Leveraging Parallel I/O in OmpSs-2

Autor
Aleix Sanchis Ramírez

Director
Vicenç Beltran Querol (BSC-CNS)

Ponent
Juan Jose Costa Prats (DAC)

Grau en Enginyeria Informàtica

Especialitat
Enginyeria de Computadors

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) -
BarcelonaTech

3 de Juliol de 2019

Agraïments

Vull agraïr al Juanjo i al Vicenç per donar-me la oportunitat de treballar en aquest projecte i agafar un tastet impagable del que suposa fer investigació, per guiar-me en els dubtes que sorgien al llarg del camí i per empènyem a millorar continuament tota la feina feta.

Agraïr al Kevin la seva paciència infinita per ajudar-me a depurar tots els problemes d'implementació que han anat sorgint al llarg del projecte.

Al Rodrigo per ensenyar-me a fer les coses ben fetes i utilitzar les *coreutils* de forma eficient.

Al Simone, Toni, Marc i altres companys d'oficina per fer més lleugers els moments difícils.

Resum

Aquest document mostra el desenvolupament portat a terme per a crear una capa dins del model de programació OmpSs-2 que permeti la interoperabilitat entre SPDK i el runtime Nanos6. Es començarà dissenyant una llibreria anomenada TASPDK (*Task-Aware* SPDK) que proporcionarà una interfície més còmoda per al programador oferint un accés a blocs de dispositius de forma transparent a SPDK. Després es crearan programes de prova, i s'avaluarà el rendiment mitjançant un *External Mergesort*, un *K-Means clustering* i un *Convolutional Image Filter*.

Resumen

Este documento muestra el desarollo realizado para crear una capa dentro del modelo de programación OmpSs-2 que permita la interoperabilidad entre SPDK i el *runtime* Nanos6. Se empezará diseñando una libreria llamada TASPDK (*Task-Aware* SPDK) que proporcionará una interfaz más cómoda para el programador que ofrezca un acceso a bloques de dispositivos de forma transparente a SPDK. Después se crearán programas de prueba, i se medirá el rendimiento a través de un *External Mergesort*, un *K-Means clustering* y un *Convolutional Image Filter*.

Abstract

This document shows the development done in order to create a layer within the OmpSs-2 programming model that allows the interoperability between SPDK and the runtime Nanos6. First, a library called TASPDK (*Task-Aware* SPDK) will be created, which will provide developers a convenient interface to devices' blocks in a transparent manner to SPDK. Then, some benchmarks will be implemented in order to evaluate the performance, such as an External MergeSort, a K-Means clustering and a Convolutional Image Filter.

Índex

1 Context **1**
- 1.1 Introducció . . . 1
- 1.2 Actors Implicats . . . 2

2 Estat de l'art **3**
- 2.1 OmpSs-2 . . . 3
  - 2.1.1 Definició . . . 3
  - 2.1.2 Funcionament i model d'execució . . . 4
  - 2.1.3 Entorn . . . 4
- 2.2 *Non Volatile Memory Express* (NVMe) . . . 5
- 2.3 SPDK . . . 8

3 Abast del projecte **10**
- 3.1 Motivació . . . 10
- 3.2 Objectius . . . 10
- 3.3 Requeriments . . . 10
- 3.4 Riscs i possibles Solucions . . . 10
  - 3.4.1 Fallada de l'equip de desenvolupament . . . 10
  - 3.4.2 Problemes de planificació . . . 11
  - 3.4.3 No disponibilitat de recursos de computació . . . 11
  - 3.4.4 Errors d'implementació . . . 11
  - 3.4.5 Problemes amb OmpSs-2 o SPDK . . . 11

4 Metodologia **12**
- 4.1 Mètodes de treball . . . 12
- 4.2 Eines de validació . . . 12
- 4.3 Eines de seguiment . . . 12
- 4.4 Avaluació del resultat final . . . 12

5 Planificació temporal **13**
- 5.1 Descripció de les tasques . . . 13
  - 5.1.1 Gestió de Projectes (GEP) . . . 13
  - 5.1.2 Estudi del *Programming Model* i del *runtime* de Nanos6 . . . 14
  - 5.1.3 Estudi de la llibreria SPDK . . . 14
  - 5.1.4 Plantejament general del projecte . . . 14
  - 5.1.5 Preparació de l'entorn de desenvolupament . . . 14
  - 5.1.6 Integració d'SPDK amb Nanos6 i OmpSs-2 . . . 14
  - 5.1.7 Creació de programes de prova i avaluació de rendiment . . . 15
  - 5.1.8 Redacció de la Memòria final . . . 15
- 5.2 Dependències entre tasques . . . 15
- 5.3 Previsió temporal . . . 16
- 5.4 Recursos . . . 16
  - 5.4.1 Recursos Hardware . . . 16
  - 5.4.2 Recursos Software . . . 17
  - 5.4.3 Recursos Humans . . . 17
- 5.5 Valoració d'alternatives i pla d'acció . . . 17

6 Gestió Econòmica **18**
- 6.1 Gestió econòmica . . . 18

6.1.1 Pressupost . . . 18
6.1.2 Control de Gestió . . . 20
6.2 Desviacions respecte a la planificació final . . . 20
6.2.1 Desviacions temporals . . . 21
6.2.2 Desviacions en el Pressupost . . . 21

7 Sostenibilitat i compromís social **23**
7.1 Dimensió Econòmica . . . 23
7.2 Dimensió Ambiental . . . 23
7.3 Dimensió Social . . . 24

8 *Task Aware* SPDK **25**
8.1 Objectiu . . . 25
8.2 Requisits . . . 27
8.2.1 *Pause/Resume* API . . . 27
8.2.2 API d'Events Externs . . . 28
8.2.3 *Polling Services* API . . . 28
8.3 Interfície de TASPDK . . . 29
8.3.1 Inicialització . . . 29
8.3.2 Creació de fitxers . . . 30
8.3.3 Lectures i Escritures . . . 31

9 Resultats i Avaluació **33**
9.1 Metodologia . . . 33
9.2 *Overhead* respecte a SPDK . . . 33
9.3 *External* MergeSort . . . 33
9.3.1 *Strong Scalability* . . . 37
9.3.2 *Bandwidth* . . . 38
9.4 *K-Means* . . . 41
9.4.1 *Strong Scalability* . . . 43
9.5 *Convolutional Image Filter* . . . 48
9.5.1 *Strong Scalability* . . . 51
9.5.2 *Bandwidth* . . . 52

10 Conclusions **56**

11 Treball futur **56**

Apèndixs **58**

A Diagrama de Gantt **58**

Índex de figures

1 *MergeSort en C amb directives OmpSs* . . . 3
2 *Esquemes dels models d'execució Thread-Pool i Fork-Join* . . . 4
3 *Entorn del model OmpSs* . . . 5
4 Stack hardware i software SATA i NVMe . . . 6
5 Esquema de la *Multi-Queue Block Layer* . . . 7
6 Model emprat que reserva una parella de cues per core de CPU . . . 7
7 Conjunt d'elements que formen SPDK . . . 8
8 Capçaleres d'algunes funcions d'SPDK utilitzades en aquest projecte . . . 9
9 *Exemple d'ús del punter virtual* . . . 25
10 Codi necessari per llegir un bloc de dades utilitzant SPDK . . . 26
11 Codi necessari per llegir un bloc de dades utilitzant SPDK . . . 27
12 *Capçaleres de les crides de la API Pause/Resume necessàries* . . . 28
13 *Capçaleres de les crides de la API d'events externs necessàries* . . . 28
14 *Capçaleres de les crides de la API de Polling Services necessaries* . . . 29
15 *Capçaleres de les crides d'inicialització i finalització de TASPDK* . . . 29
16 *Capçaleres de les crides TASPDK per crear fitxers* . . . 30
17 *Procés de creació d'un fitxer TASPDK* . . . 30
18 *Capçaleres de les crides per fer lectures, escriptures i reserva de memòria* . . . 31
19 *Exemple de la interacció de la clàusula final i les crides asíncrones* . . . 32
20 Comparació de l'ample de banda màxim assolible entre SPDK i TASPDK . . . 34
21 Codi que demostra la creació de tasques recursives . . . 35
22 Codi que demostra la sincronització entre tasques dins de la operació de merge . 36
23 Strong scaling de l'*External* MergeSort amb un fitxer d'1GB per diverses mides de bloc . . . 37
24 Eficiència de l'*External* MergeSort amb un fitxer d'1GB per diverses mides de bloc 38
25 *Bandwidth* obtingut en funció del número de cores i en funció de la mida de bloc per un fitxer d'1GB . . . 39
26 Temps total mesurat en funció del número de cores i en funció de la mida de bloc per un fitxer d'1GB . . . 39
27 Traça d'Extrae d'una execució de l'*External Mergesort* . . . 40
28 Visualització d'un *k-means clustering* de 256 punts en 5 clústers . . . 41
29 Graf de tasques creat en una iteració del *k-means* . . . 42
30 Codi que s'encarrega de la fase d'assignació . . . 43
31 *Speedup* obtingut per a diferents mides de bloc . . . 44
32 *Bandwidth* aconseguit per a diferents mides de bloc i número de *cores* . . . 44
33 Fracció de temps respecte al temps total de la part computacional. Execució amb una mida de bloc de 8MB . . . 45
34 *Speedup* aconseguit en la part computacional per diferents mides de bloc . . . 45
35 Comprovació de la convergència a través d'*unrolling* . . . 46
36 Iteracions del *k-means* utilitzant un *taskwait* . . . 47
37 Iteracions del *k-means* utilitzant l'*unrolling* . . . 47
38 Graf de tasques creat per filtrar un bloc de la imatge. . . . 49
39 Codi que s'encarrega d'implementar la convolució paral·lelitzat mitjançant tasques 50
40 Speedup utilitzant diferents mides de bloc . . . 51
41 Traces d'Extrae d'una execució utilitzant les crides síncrones . . . 52
42 Codi encarregat de paral·lelitzar la lectura de fragments de la imatge des de disc 52
43 Bandwidth utilitzant diferents mides de bloc . . . 53
44 Anàlisi de la fracció de temps que es dedica a tasques de còmput utilitzant una mida de bloc de 0.25MB . . . 54

45 *Bandwidth* obtingut en funció de la mida de bloc per a 56 cores 54
46 *Speedup aconseguit en la convolució* 54
47 Eficiència aconseguida en la convolució 55
48 Diagrama de Gantt del Projecte 58

Índex de taules

1 *Dependències entre tasques del projecte* 15
2 *Previsió temporal de cada etapa* 16
3 Pressupost detallat del projecte 20
4 *Diferències entre les dates de finalització previstes i les dates de finalització finals* 22
5 Diferents *kernels* i els seus efectes sobre una imatge. Imatges per Michael Plotke sota llicència CC BY-SA 3.0 49

1 Context

La fracció de temps d'un programa de l'àmbit de High-Performance Computing (HPC) que està dedicada a les operacions d'Entrada/Sortida (ES), tot i que pugui semblar que no és crítica, no és despreciable.

Tradicionalment, els dispositius emprats han sigut del tipus electromecànic, com els *Hard Drive Disks* (HDD) que funcionen a base de discs en rotació. Avui dia, però, cada cop més s'utilitzen dispositius d'estat sòlid (SSD) que gaudeixen d'un millor rendiment i consum energètic.

Degut a que el rendiment dels SSD és, com a mínim, un ordre de magnitud major que el dels HDD[1], el software encarregat de les transaccions és cada cop més crític, i alguns sistemes, per exemple el protocol SATA, o la *stack* d'E/S de Linux no estan preparats per aprofitar aquesta millora dels nous dispositius ja que van ser dissenyats per a discs durs rotacionals, esdevenint un coll d'ampolla[2].

En les següents subseccions s'introdueix aquest projecte i s'especifica el personal involucrat en el desenvolupament d'aquest.

1.1 Introducció

Aquest projecte és un Treball Final de Grau (TFG) del Grau d'Enginyeria Informàtica (GEI) de la Facultat d'Informàtica de Barcelona (FIB), emmarcat dins de l'especialitat d'Enginyeria de Computadors. Ha estat desenvolupat en col·laboració amb el Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS).

OmpSs-2 és un *programming model* de memòria compartida basat en tasques i influenciat per dos altres models, OpenMP i StarSs[3]. Ha estat desenvolupat al BSC, i, com OpenMP, funciona anotant el codi font amb diverses directives, que permeten al compilador generar un programa paral·lel. Ofereix suport al paral·lelisme asíncron i a processadors heterogenis com GPUs o acceleradors. La implementació de referència d'OmpSs-2 està basada en el compilador *source-to-source* Mercurium[4] i el runtime Nanos6[5]. Més endavant s'entra en detall en tots els components que formen l'entorn del model d'OmpSs-2.

El protocol *Non Volatile Memory Express* (NVMe) és un protocol destinat a discs durs d'estat sòlid que permet explotar les característiques d'aquests dispositius, com ara el paral·lelisme intern o la baixa latència[6].

SPDK és un conjunt d'eines desenvolupades per Intel amb la finalitat d'explotar al màxim el rendiment dels dispositius NVMe intentant solucionar el coll d'ampolla que presenta el *software* que està actualmentent en ús[7].

La finalitat d'aquest projecte és desenvolupar una llibreria que s'integri dins del *programming model* OmpSs-2 per explotar el rendiment dels discs NVMe, utilitzant SPDK però encapsulant-ne el funcionament per tal d'oferir una interfície fàcil d'usar similar a les existents per manipular fitxers. Així es dona una via a desenvolupadors per explotar al màxim el paral·lelisme de les seves aplicacions que facin un ús intensiu d'operacions d'E/S.

1.2 Actors Implicats

El desenvolupament d'aquest projecte té una sèrie de persones implicades, detallades a continuació:

- **Desenvolupador**: En aquest cas, jo mateix. És la persona encarregada de realitzar aquest projecte. La seva feina inclou anàlisi dels requisits, investigació, desenvolupament i verificació i obtenció de resultats, incloent també la creació de documentació respecte al projecte.

- **Director i Ponent**: Seran les persones encarregades de supervisar, guiar i assegurar que el desenvolupador assoleix les fites marcades, a través de reunions periòdiques.

- **Barcelona Supercomputing Center**: El centre subministrarà al desenvolupador el material necessari per dur a terme la seva tasca, com ordinador o pantalles, i recursos de computació per tal d'obtenir resultats de rendiment en un entorn real.

- **Beneficiaris**: Potencialment aquest projecte aportarà una eina a usuaris del *programming model* OmpSs-2 per tal d'accelerar els seus programes en l'aspecte d'entrada/sortida, per a ús tant en HPC com en Big Data.

2 Estat de l'art

En aquest apartat es tracta l'estat de l'art i la contextualització, on s'explica de forma detallada el model de programació d'OmpSs-2, el protocol NVMe i la llibreria SPDK

2.1 OmpSs-2

2.1.1 Definició

OmpSs-2[3] és un model de programació paral·lela, creat pel grup de *Programming Models* del departament de Computer Science del Barcelona Supercomputing Center (BSC), que estén OpenMP amb conceptes del model StarSs, també desenvolupat al BSC. Dóna noves característiques a OpenMP com ara paral·lelisme asíncron a través de dependències de dades i dispositius heterogenis. OmpSs-2 està disponible per C/C++ i Fortran.

L'objectiu d'OmpSs-2 és oferir un entorn productiu al voltant del qual es puguin desenvolupar aplicacions per a sistemes de computació d'altes prestacions (HPC). Els programes basats en OmpSs-2 haurien de presentar un rendiment comparable amb altres models de programació que explotin les mateixes arquitectures. A més, OmpSs-2 intenta oferir aquest rendiment mantenint una senzillesa d'ús.

Respecte a l'últim punt esmentat, OmpSs-2 hereda d'OpenMP la forma de paral·lelitzar el codi font, a través de directives. L'usuari annota el codi seqüencial, però sense la necessitat d'haver de paral·lelitzar zones de codi explícitament. Aquest enfocament permet també una paral·lelització incremental, on es van paral·lelitzant diferents parts de l'aplicació. La figura 1 mostra un exemple on es pot apreciar la senzillesa d'ús del model de programació.

OmpSs-2 dóna també suport a l'heterogeneïtat de dispositius com *GPUs*. Per heterogeneïtat de dispositius s'entén els sistemes que usen més d'un tipus de processador. Aquests sistemes guanyen en rendiment no només per integrar un sol tipus de processador sinó coprocessadors amb capacitats de processament especialitzades per a cert tipus de tasques.

```
void merge_sort(int *a, int l, int r) {

    if(l < r){

        int m = (l+r)/2;

        #pragma oss task
        merge_sort(a, l, m);

        #pragma oss task
        merge_sort(a, m, r);

        #pragma oss taskwait
        merge(a, l, m, r);
    }

}
```

Figura 1: *MergeSort en C amb directives OmpSs*

2.1.2 Funcionament i model d'execució

La unitat bàsica d'execució en OmpSs-2 és la tasca. Tot el model d'OmpSs-2 està basat en tasques i les dependències que presenten entre elles. Les dependències permeten al programador especificar el flux de dades del programa, i el *runtime* és l'encarregat de fer servir aquesta informació per determinar si l'execució paral·lela de dues tasques presentaria conflictes en l'ús de les dades.

El model d'execució d'OmpSs-2 és del tipus *thread-pool*. El *runtime*, al principi de l'execució, crea un conjunt de *threads*, fils d'execució. Aquests *threads* seran els encarregats d'executar les diverses tasques que composin el programa. Hi ha dos tipus de *threads*, el master, que executa el codi de forma seqüencial de tot el programa, i la resta, que esperen a que hi hagi tasques disponibles per executar-se. Cal destacar que existeix una tasca implícita, que engloba tot el codi.

Quan un *thread*, master o no, es troba una clàusula de tasca, es genera una tasca explícita, i s'associa a un dels *threads* disponibles. Si la tasca no té cap dependència per resoldre, i el *thread* està disponible, s'executarà de forma immediata. Si no, esperarà a que es compleixin ambdues condicions.

Aquest model dista del usat en OpenMP, que és del tipus *fork-join*. En el *fork-join*, el codi s'executa de forma seqüencial fins a arribar a una regió paral·lela, anotada per l'usuari. Allà es creen la resta de threads, que executen el codi de forma paral·lela, i s'agrupen al final d'aquesta. Cada una d'aquestes regions paral·leles poden ser dividides recursivament fins a assolir una granularitat desitjada. La figura 2 il·lustra la diferència entre els dos models mencionats.

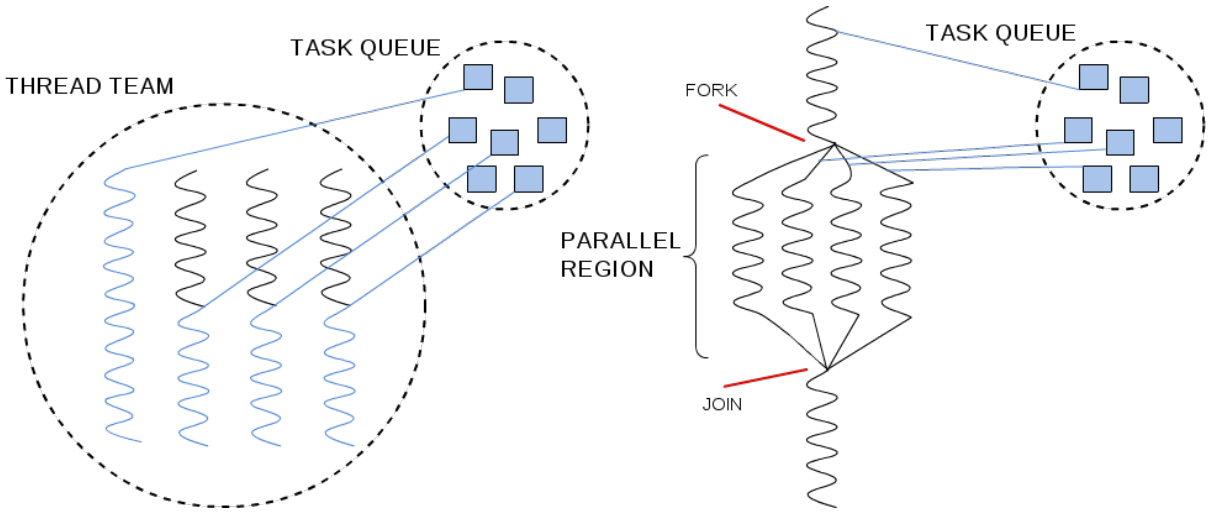


Figura 2: *Esquemes dels models d'execució Thread-Pool i Fork-Join*

2.1.3 Entorn

OmpSs-2 està format per un conjunt d'eines. La implementació de referència consta del compilador Mercurium[4], el *runtime* Nanos6[5], i una sèrie d'eines d'anàlisi, entre les que destaquen *Paraver* [8] i *Extrae* [9]. L'ecosistema complet d'OmpSs-2 es pot observar a la figura 3.

- **Mercurium** és una infraestructura de compilació *source to source*, que generalment s'usa

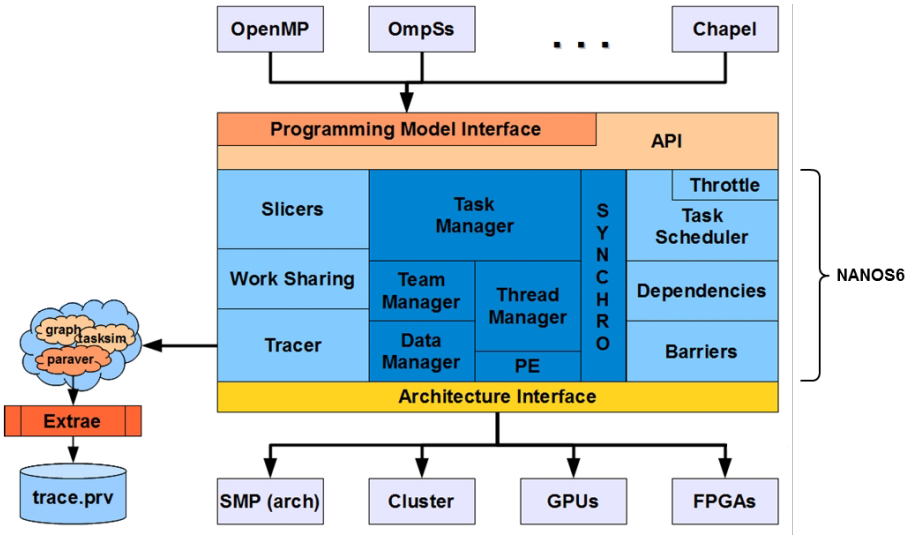


Figura 3: *Entorn del model OmpSs*

per implementar OpenMP en l'entorn del runtime de *Nanos*, però que suporta més models. Està disponible pels llenguatges *C, C++* i *Fortran*.

- **Nanos6** és una llibreria designada per a ser usada com a *runtime* de suport a entorns paral·lels. Normalment s'usa per donar suport al model OmpSs-2, però també té la capacitat per suportar OpenMp i Chapel [10]. Nanos6 proporciona serveis per suportar paral·lelisme a nivell de tasques que es sincronitzen mitjançant dependències de dades o a través de sincronitzacions explícites. A més, ofereix suport per mantenir la coherència entre diferents espais d'adreces, com per exemple amb GPUs i nodes de clúster.

- ***Paraver*** és una eina que permet veure traces d'execució de programes paral·lels, a traves de diferents vistes o configuracions i utilitzant inspeccions visuals, de forma que l'usuari pugui veure amb exactitud el comportament de les aplicacions.

- ***Extrae*** és el *package* encarregat de generar traces de *Paraver* per tal de realitzar un anàlisi posterior a l'execució. Funciona a traves de sondes que s'insereixen en el programa i permet una customització d'events personalitzats.

2.2 *Non Volatile Memory Express* (NVMe)

Non Volatile Memory Express, d'ara endavant NVMe, és una especificació d'interfície *host-controller* dissenyada per accedir a memòria d'emmagatzematge no volàtil a través del bus *PCI Express* (PCIe) [11]. Aquesta interfície ha estat dissenyada des de zero per tal d'aprofitar al màxim la baixa latència i el paral·lelisme intern en els dispositius d'emmagatzematge d'estat sòlid.

La necessitat per aquest nou protocol neix de la poca adequació que presentaven els busos usats fins al moment, com ara SATA o SAS. El bus SATA s'havia convertit en la forma més habitual de connectar dispositius SSD als ordinadors personals, però havia estat dissenyat per interactuar amb discs durs mecànics, HDDs, i va començar a esdevenir un coll d'ampolla per a molts SSDs [12]. Ja existien dipositius d'alta gamma que utilitzaven el bus PCIe, però utilitzaven protocols no estàndar. Amb la creació d'NVMe, els sistemes operatius només necessiten un *driver* per tal de funcionar amb tots els SSDs que s'adhereixin a la especificació. La versió 1.0 d'aquest

protocol va ser publicada l'1 de Març de 2011, i va ser desenvolupada per un grup format per més de 90 companyies. El kernel de Linux 3.3, publicat el 19 de Març de 2012 va incloure el driver NVMe [13].

Els avantatges de rendiment que presenta NVMe sobre SATA emanen principalment de tres aspectes:

- **Capacitat de l'interfície**: El bus PCIe permet un *throughput* molt superior al del bus SATA. SATA 3, la versió més actual, ofereix un màxim de 600MB/s. Per contra, una sola línia PCIe permet un *bandwidth* de 1GB/s [12]. Els SSDs moderns típicament són dipositius PCIe de tercera generació x4, per tant suporten fins a 4GB/s. Els *chipsets* actuals de processadors x86 tenen *slots* PCIe x8 i x16, per tant permeten *bandwidths* de 8 i 16 GB/s, respectivament. Així doncs, aquesta interfície està limitada només pel número màxim de línies PCIe suportades pel microprocessador.

- ***Datapath Hardware***: A la figura 4, part a) es pot observar la diferència entre ambdós *datapath*. Un dispositiu SATA típicament es connecta al sistema a través d'un adaptador de bus del host (HBA). Una petició d'E/S en un sistema x86, per tant, ha de passar pel driver AHCI, el bus del host, l'adaptador del bus del host i finalment al dispositiu. En canvi, un dispositiu NVMe es connecta directament al sistema a través del *PCIe root complex port*. Aquest *datapath* més curt ajuda a reduïr substancialment la latència [12].

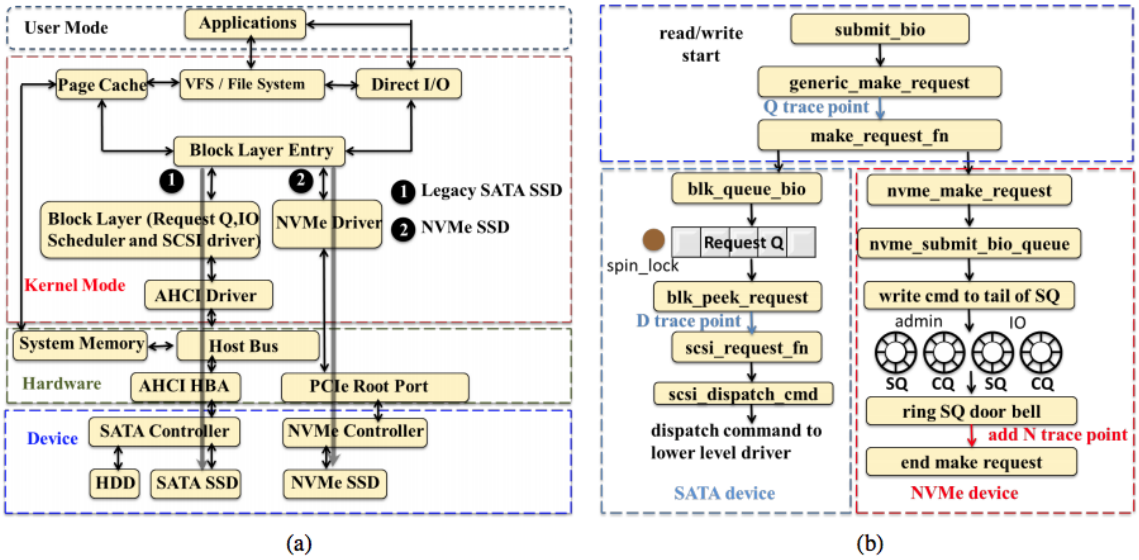


Figura 4: Stack hardware i software SATA i NVMe

- **Software Stack**: A la figura 4, part b) es pot observar la *stack software* per un dispositiu SATA i per un NVMe. En el camí tradicional d'E/S, una petició que arriba a la *block layer* serà afegida a una cua *software* de peticions. En aquesta cua, anomenada *Elevator*, les peticions són combinades i reordenades per tal de convertir múltiples peticions en una petició seqüencial. Aquesta optimització, molt necessària en dispositius HDD, degut al seu baix rendiment en accessos aleatoris, és redundant en SSDs, ja que presenten gairebé la mateixa latència en accessos seqüencials com en accessos aleatoris. Aquesta cua presenta un punt de contenció problemàtic en arquitectures paral·leles i augmenta la latència [12].

  Per tal de paliar aquests problemes, la *stack software* NVMe de Linux inicialment va optar per fer un *bypass* del kernel, però actualment el kernel disposa de la *Multi-Queue Block Layer*[14], que tot i que elimina el punt de contenció que és l'Elevator utilitzant

diverses cues *software*, no elimina l'*overhead* del canvi de context al *kernel*. Es pot veure un esquema de l'arquitectura d'aquesta capa a la figura 5.

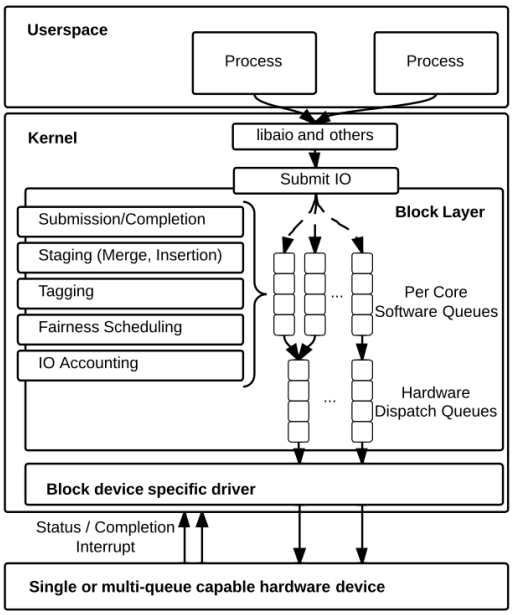


Figura 5: Esquema de la *Multi-Queue Block Layer*

L'estàndard NVMe implementa un mecanisme basat en cues *hardware* emparellades, de *Submission* i de *Completion*. NVMe suporta fins a 64K cues d'E/S i fins a 64K comandes per cua. A la pràctica, però, els dipositius suporten fins un màxim de 128 cues [15].

Aquestes cues resideixen en memòria *host* i són manipulades de forma cooperativa entre el driver NVMe i el controlador NVMe; les peticions són afegides a la cua de *submission* pel driver i és el controlador qui agafa aquestes peticions, les processa, i afegeix entrades a la cua de *completions* quan les peticions són completades. Típicament es crea una parella de cues per cada *core* de la CPU, eliminant així el punt de contenció que presentava l'*Elevator*. A la figura 6 es pot observar un exemple d'aquest paradigma.

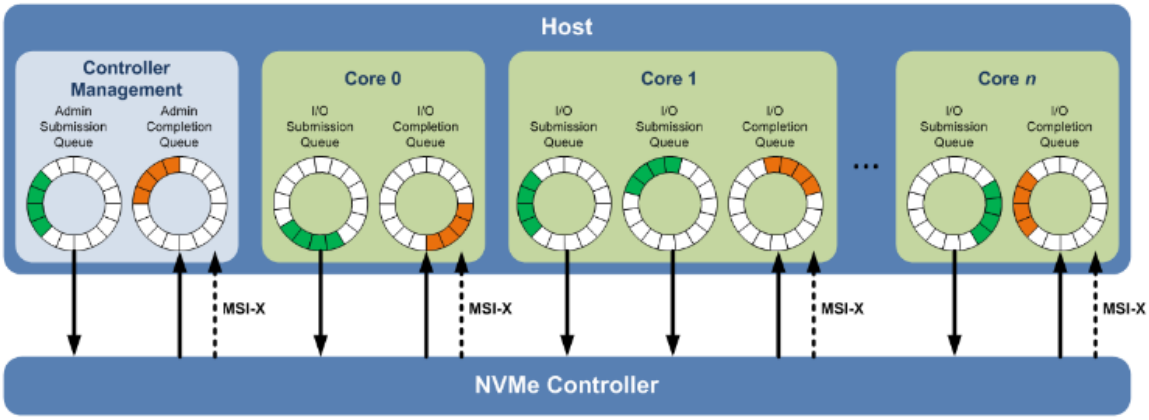


Figura 6: Model emprat que reserva una parella de cues per core de CPU

2.3 SPDK

L'*Storage Performance Development Kit*, d'ara endavant SPDK, és una col·lecció d'eines i llibreries desenvolupades per Intel pensades per usar-se en aplicacions basades en *storage*, que siguin escalables, i que necessitin un alt rendiment [2]. Per exemple, recentment, es va aconseguir 10.39 milions d'operacions per segon (IOPS) utilitzant un sol thread a través d'SPDK [16].

La base d'SPDK, i el component utilitzat en aquest projecte, és un driver NVMe que funciona en *user-space*, per *polling*, asíncron i *lockless*. Això permet un accés directe i altament paral·lel a un SSD des de una aplicació en *user-space*, per tant evitant còpies a memòria de *kernel* i canvis de context. SPDK també proporciona una *block stack* similar a la que ofereix un sistema operatiu, o suport per a NVMe Over Fabrics (NVMeOF), que permet comunicació entre nodes i dispositius d'emmagatzematge a través d'una xarxa. A la figura 7 es mostra el conjunt d'eines que formen SPDK.

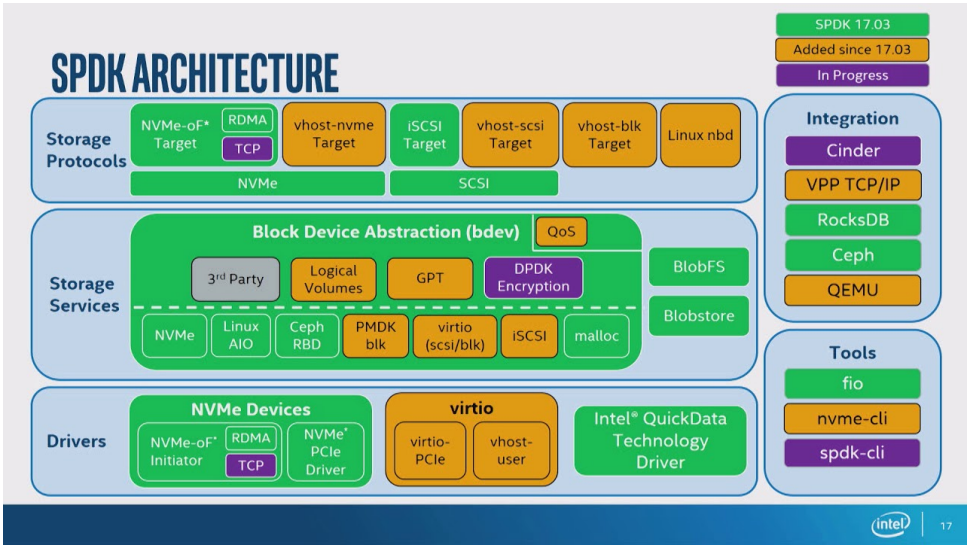


Figura 7: Conjunt d'elements que formen SPDK

La motivació darrere d'SPDK és maximitzar l'eficiència i el rendiment dels dispositius NVMe d'última generació, a través de minimitzar l'impacte del *software* encarregat de les transaccions d'E/S. Per tal d'aconseguir-ho s'utilitzen principalment dues tècniques:

- **Funcionament en *user-space*:** Generalment els drivers, software que s'encarrega d'interactuar amb els dispositius, s'executen en espai de memòria del *kernel*, ja que requereixen accés a instruccions i adreços especials per tal de realitzar les operacions d'E/S. El problema d'aquest mètode és que cada operació requereix un canvi de context i còpia de dades entre el procés d'usuari i el *kernel*. Si es realitza un número elevat d'operacions per segon, existeix un *overhead* important de cicles que es perden en aquests canvis. Si més d'una aplicació no ha de compartir el dispositiu, una alternativa és els drivers en *user-space*. Aquests drivers s'executen en nivell de privilegi d'usuari, i per tant s'eviten els canvis de context, però tot i així disposen d'accés directe al hardware amb el qual interaccionen.

- ***Polling*:** L'estratègia tradicional per tal d'efectuar operacions d'E/S era enviar la petició i programar una interrupció per tal de ser notificats quan s'hagi completat. Aquesta estratègia és òptima quan s'espera una latència elevada entre la petició i l'obtenció de resultats, com succeeix en els dispositius mecànics. En el cas dels dipositius NVMe, però, l'*overhead* resultant de fer un canvi de context al *kernel* i programar la interrupció acaba

esdevenint comparable a la latència del propi dispositiu[17]. Per això, SPDK utilitza una estratègia basada en *polling*, on només és necessari comprovar un bit que resideix en memòria del *host* [18].

Per tal que SPDK pugui utilitzar un dispositiu NVMe, s'ha de deslligar el driver del *kernel* que està carregat, i utilitzar el driver genèric *vfio*. Aquest driver és un driver que informa al *kernel* que aquell dispositiu ja està gestionat per un driver en *user-space* per tal d'evitar que l'hi assigni el driver per defecte.

SPDK, com tots els drivers NVMe, funciona per un mecanisme anomenat Direct memory access (DMA). DMA permet que dispositius hardware accedeixin a memòria RAM sense intervenció de la CPU, per tal que aquesta pugui fer altres tasques mentre es completa la operació de transferència. Per fer operacions de DMA en *user-space*, SPDK utilitza el mecanisme de *hugepages*[19]. En la distribució d'SPDK es proporciona un script que reserva un número de pàgines configurable i deslliga el driver del dispositiu pel driver *vfio*. És necessari executar-lo per configurar el sistema abans de poder utilitzar SPDK.

Totes les crides per efectuar E/S al driver NVMe d'SPDK segueixen la nomenclatura *nvme_ns_cmd_xxx*, on *xxx* determina la operació que es vol efectuar, com ara *read* o *write*. Totes les funcions d'E/S d'SPDK retornen immediatament sense esperar que es completin. En canvi, accepten un punter a una funció que es cridarà quan la petició estigui completa. A la figura es mostren les capçaleres d'algunes crides utilitzades per implementar TASPDK. A la web `https://spdk.io/doc/nvme_8h.html` es pot consultar el conjunt de crides que ofereix el driver NVMe d'SPDK i una explicació més detallada del seu comportament.

```
/* Crida per efectuar operacions d'escriptura. La crida de read és idèntica.
* lba determina el número de bloc a partir del qual s'inicia la petició
* lba_count indica el número de blocs que es llegiran
* cb_fn és una funció que es cridarà quan es completi la operació
* cb_arg és un punter a dades que es poden passar a cb_fn
*/
int spdk_nvme_ns_cmd_write  (struct spdk_nvme_ns *ns, struct spdk_nvme_qpair *
    qpair, void *payload, uint64_t lba, uint32_t lba_count, spdk_nvme_cmd_cb cb_fn
    , void *cb_arg, uint32_t io_flags);

/* Crida per crear una parella de cues NVMe
*/
struct spdk_nvme_qpair* spdk_nvme_ctrlr_alloc_io_qpair  (struct spdk_nvme_ctrlr *
    ctrlr, const struct spdk_nvme_io_qpair_opts *opts, size_t opts_size)

/* Crida per fer polling a una cua NVMe per comprovar quines peticions han acabat.
*/
int32_t spdk_nvme_qpair_process_completions (struct spdk_nvme_qpair *qpair,
    uint32_t max_completions)
```

Figura 8: Capçaleres d'algunes funcions d'SPDK utilitzades en aquest projecte

3 Abast del projecte

En aquest apartat es tractarà l'abast del projecte, els seus objectius i la manera d'assolir-los. Tot això inclou la motivació darrere del projecte, la metodologia de treball emprada, els requeriments i els possibles riscos que cal preveure.

3.1 Motivació

Tal com s'ha mencionat anteriorment, la motivació principal és desenvolupar una llibreria integrada dins del *programming model* OmpSs-2 que permeti explotar el rendiment dels dispositius NVMe utilitzant SPDK, però encapsulant-ne els detalls i oferint una interfície còmoda d'usar.

3.2 Objectius

- Aconseguir millores de rendiment substancials en programes HPC que usin el model de programació OmpSs-2 i que facin un ús intensiu d'operacions d'E/S.
- Emprar bones pràctiques de programació, amb un estil net i de baixa complexitat.

3.3 Requeriments

- Les eines desenvolupades han de ser eficients, fàcils d'usar i permetre una fàcil integració a nous projectes.
- Mantenir una constant comunicació bidireccional amb diferents departaments del BSC-CNS per tal de suggerir millores que podrien sorgir a partir del seu ús.

3.4 Riscs i possibles Solucions

Durant el desenvolupament poden sorgir problemes, i per tal de millorar el nostre temps de reacció, intentarem llistar possibles riscs i les seves solucions.

3.4.1 Fallada de l'equip de desenvolupament

En cas de fallada de l'equip usat per desenvolupar la eina, o d'algun perifèric com pantalla o teclat.

Solució: És important mantenir backups ja que, en cas d'una fallada general del sistema, no es perdria feina feta i l'únic problema seria el *downtime* a la espera de rebre un nou equip.

3.4.2 Problemes de planificació

En algun punt durant el desenvolupament el desenvolupador s'endarrereixi en la realització d'alguna tasca, per diferents motius.

Solució: Realitzar una planificació honesta i curosa, preveient marges suficients per tal de paliar aquests problemes.

3.4.3 No disponibilitat de recursos de computació

Les proves i les obtencions de resultats s'hauran de realitzar en un entorn real de computació. Es podria donar el cas que aquestes màquines no funcionin o que hi hagi altres usuaris competint per ells.

Solució: Tot i que aquesta situació es pot donar, no és habitual. Si això passa, mentrestant el desenvolupador es pot dedicar a altres tasques com manteniment de codi, o escriure documentació.

3.4.4 Errors d'implementació

Tot i que és inevitable realitzar errors durant el desenvolupament de qualsevol projecte, el desenvolupador es podria trobar encallat en alguna etapa per un error difícil d'arreglar.

Solució: Caldrà utilitzar eines de *debug* de forma correcta per tal de perdre el menor temps arreglant errors, i planificar marges per aquestes situacions.

3.4.5 Problemes amb OmpSs-2 o SPDK

SPDK és un projecte en constant desenvolupament i poden aparèixer errors que caldrà identificar correctament, el mateix podria passar amb qualsevol eina de l'ecosistema d'OmpSs-2.

Solució: Mantenir una comunicació amb l'equip de desenvolupament d'SPDK a través d'eines telemàtiques. En el cas d'OmpSs-2, serà més fàcil ja que l'eina es desenvolupa dins del BSC-CNS i hi ha moltes persones dins l'equip amb un alt nivell d'experiència amb ella.

4 Metodologia

En aquesta secció s'especificaran els mètodes de treball usats en el projecte i les eines de seguiment emprades per tal de garantir-los.

4.1 Mètodes de treball

Com que el desenvolupador serà l'única persona treballant en el projecte de forma directa, s'utilitzarà la metodologia *SCRUM*[20]. En aquesta metodologia, el desenvolupament és incremental i basat en iteracions de petits espais de temps. Aquest mètode permet canviar la planificació a mesura que avança el projecte, idea molt important en un treball d'investigació. Es consultarà l'estat del treball periòdicament, avaluant el progrés fet en la última iteració, predint el progrés que s'assolirà per la pròxima, i quins impediments podrien evitar-ho.

4.2 Eines de validació

És important establir una sèrie de tests quan s'afronta un projecte *software* de certa complexitat, per tal de detectar errors amb suficient previsió i que no comprometin la planificació sencera del projecte. Durant el desenvolupament de la llibreria es duran a terme tests per a cada component que la forma, així com tests d'integració que asseguraran que funcionen correctament entre ells.

Les reunions periòdiques amb el director del projecte també ajudaran a detectar errors comesos ja que serviran com a segona fase de revisió per a detalls que puguessin passar per alt.

4.3 Eines de seguiment

Durant tot el projecte el desenvolupador es mantindrà en contacte amb el Director a través d'eines com correu electrònic, *Slack* i de reunions periòdiques (com a mínim setmanals). També s'usaran eines de control de versions com *Git* i un repositori online al *Gitlab* del departament de *Programming Models* del BSC.

4.4 Avaluació del resultat final

Un cop la llibreria estigui acabada i sigui funcional, es desenvoluparan programes de prova que l'utilitzin, i es medirà el seu rendiment per tal de veure si coincideix amb l'esperat, i, en cas contrari, les causes que ho provoquen per tal de buscar possibles solucions.

5 Planificació temporal

En aquesta secció es detallarà la forma en que s'organitzarà el temps i les tasques a realitzar per a completar el projecte, la duració de cada una i les dependències que presentin entre elles. S'inclourà un diagrama de Gantt, s'especificaran els recursos necessaris per a dur les tasques a terme, i els possibles desviaments temporals que es puguin produir, i els seus possibles efectes.

5.1 Descripció de les tasques

A continuació es detallen les tasques que es realitzaran durant el projecte. S'ha dividit entre 8 tasques, que figuren a continuació. En total el projecte requerirà 545 hores.

- Gestió de Projectes (GEP)
- Estudi del *Programming Model* i del *runtime* de Nanos6
- Estudi de la llibreria SPDK
- Plantejament general del projecte
- Preparació de l'entorn de desenvolupament
- Integració d'SPDK amb Nanos6 i OmpSs-2
- Creació de programes de prova i avaluació de rendiment
- Redacció de la Memòria final

5.1.1 Gestió de Projectes (GEP)

La primera tasca del projecte es realitzarà en l'assignatura de GEP, i es conformarà de quatre entregues, que són les següents:

- **Abast del projecte i contextualització** 24 hores
- **Planificació temporal** 8 hores
- **Gestió econòmica i sostenibilitat** 18 hores
- **Presentació Oral i Document final** 20 hores

En total ens surt una dedicació de 70 hores. Les 5 hores restants del total teòric de 75 hores estaran dedicades a cobrir possibles imprevistos que puguin sorgir.

Per a realitzar aquesta tasca s'utilitzarà un ordinador, l'aplicatiu web *OverLeaf*, el cercador *Google*, i el software de planificació *Gantter*.

5.1.2 Estudi del *Programming Model* i del *runtime* de Nanos6

Per tal de completar aquest projecte de forma satisfactòria caldrà un coneixement profund del model de programació que s'utilitzarà i del *runtime* de Nanos6 per tal de conseguir una integració eficient i usable.

El desenvolupador haurà d'estudiar profundament la documentació disponible d'ambdós i podrà interactuar amb altres desenvolupadors que estiguin més familiaritzats amb les eines.

Per tal de realitzar aquesta tasca el desenvolupador necessitarà un ordinador amb connexió a internet.

5.1.3 Estudi de la llibreria SPDK

L'altra part integral del projecte, la llibreria SPDK, també necessitarà d'un estudi i procés de familiarització per tal d'aconseguir certa fluïdesa en el seu ús.

Els recursos necessaris per aquesta tasca son els mateixos que per la tasca anterior.

5.1.4 Plantejament general del projecte

Un cop fet un estudi de les parts que composen el projecte, s'analitzaran els objectius concrets i requeriments, que estaran coordinats amb el director del projecte. S'analitzarà quina funcionalitat es vol donar al usuari i el seu nivell d'abstracció.

Per a realitzar aquesta tasca el desenvolupador necessitarà un ordinador amb connexió a internet i algun set d'eines d'Ofimàtica.

5.1.5 Preparació de l'entorn de desenvolupament

En aquesta tasca el desenvolupador prepararà totes les eines que siguin necessàries per a completar la fase principal del projecte. Això inclou tot el software necessari, llibreries, *runtime*, compilador, editor i configuració del sistema operatiu.

Per tal de completar aquesta tasca serà necessari un ordinador amb sistema operatiu GNU/-Linux, la llibreria SPDK, el *runtime* Nanos6, el compilador Mercurium, Git, tots els paquets que siguin dependències dels esmentats, i un editor com ara *Visual Studio Code*.

5.1.6 Integració d'SPDK amb Nanos6 i OmpSs-2

Aquesta és la tasca principal del projecte i la que més temps comportarà.

Primer caldrà estudiar decidir quina interfície i quin nivell d'abstracció s'exposarà al usuari, per tal de maximitzar la productivitat però permetent cert nivell de control al programador, ja que aquesta integració ha de ser eficient.

Després caldrà implementar-la, comprovant en tot moment que no s'estan realitzant errors, especialment de gestió de memòria i de concurrència, que podrien no ser evidents d'entrada però que comportarien problemes difícils de detectar en el futur.

Per tal d'efectuar aquesta tasca el desenvolupador usarà el sistema configurat que s'ha descrit a la secció 5.1.5

5.1.7 Creació de programes de prova i avaluació de rendiment

Un cop la integració sigui completa, caldrà crear programes de prova que intentin explotar els suposats avantatges aconseguits amb la nostra implementació, avaluar el seu rendiment i comparar-lo amb implementacions actuals.

Un altre cop, per tal de poder acabar assolir aquesta tasca caldrà l'entorn configurat prèviament.

5.1.8 Redacció de la Memòria final

L'última etapa serà la de redactar la memòria del projecte i la presentació per a defensar-lo davant el tribunal.

S'utilitzarà l'editor *LaTeX Overleaf*, i LibreOffice per tal de preparar els materials audiovisuals necessaris.

5.2 Dependències entre tasques

Tasca dependent	Tasca predecessora
Gestió de Projectes	-
Estudi del *Programming Model i del runtime*	Gestió de Projectes
Estudi de la llibreria SPDK	Gestió de Projectes
Plantejament general del projecte	Estudi d'SPDK + Estudi del *Programming Model* i del *runtime*
Preparació de l'entorn de desenvolupament	Plantejament general del projecte
Integració d'SPDK amb Nanos6 i OmpSs-2	Preparació de l'entorn de desenvolupament
Programes de prova i avaluació de rendiment	Integració d'SPDK amb Nanos6 i OmpSs-2
Redacció de la memòria final	Programes de prova i avaluació de rendiment

Taula 1: *Dependències entre tasques del projecte*

5.3 Previsió temporal

A continuació figuren totes les tasques amb la seva previsió temporal. Aquesta taula complementa el diagrama de Gantt que apareix a l'apèndix A.

Tasca dependent	Número d'hores
Gestió de Projectes	75
Estudi del *Programming Model i del runtime*	30
Estudi de la llibreria SPDK	30
Plantejament general del projecte	30
Preparació de l'entorn de desenvolupament	20
Integració d'SPDK amb Nanos6 i OmpSs-2	200
Programes de prova i avaluació de rendiment	100
Redacció de la memòria final	60
Total	**545 hores**

Taula 2: *Previsió temporal de cada etapa*

5.4 Recursos

5.4.1 Recursos Hardware

- **Ordinador portàtil:** Proporcionat pel BSC-CNS, Dell Latitude 7490 8th Gen Intel® Core™ i7-8650U Processor (Quad Core, 8MB Cache, 1.9GHz,15W), 16GB, 2x8GB, DDR4 2400MHz Memory, M.2 512GB SATA Class 20 Solid State Drive.
- **Pantalla:** Proporcionada pel BSC-CNS, per millorar la comoditat del desenvolupador, Dell Professional P2715Q de 27".
- **Clústers de computació:** Proporcionats pel BSC-CNS, falta per decidir quina màquina serà assignada al projecte.

5.4.2 Recursos Software

- **Ubuntu 18.04:** El sistema operatiu utilitzat per a desenvolupar el projecte.
- **GitLab:** Per tal d'efectuar control de versions.
- **Editors:** S'utilitzarà l'editor *VsCode*.
- **Planificació:** *Gantter*.
- **Software per desenvolupar:** El compilador Mercurium, el runtime Nanos6, i el *debugger* GDB.
- **Altres:** *OverLeaf*, LibreOffice.

5.4.3 Recursos Humans

- **Director:** S'encarregarà de supervisar el projecte i guiar el desenvolupador.
- **Desenvolupador:** S'encarregarà de realitzar tot el projecte
- **Suport:** Ajudaran al Desenvolupador a realitzar tasques i l'aconsellaran.

5.5 Valoració d'alternatives i pla d'acció

Tot i que s'ha intentat ajustar el màxim la planificació a les previsions reals, sempre és difícil en un projecte d'aquest tipus preveure els problemes que ens podrem trobar en un futur. La previsió de les tasques de desenvolupament i implementació són les que més temps tenen assignades ja que són les que més problemes poden presentar.

En cas que alguna tasca es realitzés més ràpid del previst simplement es farien les següents i es podria mirar al final del projecte d'ampliar l'abast si sobra temps. En cas que alguna tasca sofrís un retràs important, es podria arribar a plantejar la cancel·lació d'aquesta. S'intentarà per totes les mesures que això no succeeixi per tal de tenir el projecte finalitzat amb els objectius plantejats a temps.

6 Gestió Econòmica

En aquesta secció es detallarà el pressupost del projecte, desglossant-lo en detall, i calculant el pressupost final del projecte.

6.1 Gestió econòmica

6.1.1 Pressupost

Per tal de descriure el pressupost s'ha optat per diferenciar entre costos directes i costos indirectes, desglossats segons cada tasca de la planificació. El pressupost apareix detallat a la següent taula

Recurs	Unitats	/u o hora ()	Vida útil (anys)	Amortització (/h)	Preu ()
Costos Directes					
Gestió de Projectes	**75 hores**				**623.25**
Dell Latitude 7490	1	1830	4	0.21	15.75
Dell Professional P2217H	1	250	4	0.03	2.25
Perifèrics	1	50	4	0.07	5.25
Ubuntu 18.04LTS	1	0	-	0	0
Overleaf	1	0	-	0	0
Gantter	1	0	-	0	0
Desenvolupador	1	8	-	-	600
Estudi del Programming Model	**30 hores**				**249.3**
Dell Latitude 7490	1	1830	4	0.21	6.3
Dell Professional P2217H	1	250	4	0.03	0.9
Perifèrics	1	50	4	0.07	2.1
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	240
Estudi de la llibreria SPDK	**30 hores**				**249.3**
Dell Latitude 7490	1	1830	4	0.21	6.3
Dell Professional P2217H	1	250	4	0.03	0.9
Perifèrics	1	50	4	0.07	2.1
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	240
Plantejament del projecte	**30 hores**				**249.3**
Dell Latitude 7490	1	1830	4	0.21	6.3
Dell Professional P2217H	1	250	4	0.03	0.9

Perifèrics	1	50	4	0.07	2.1
Ubuntu 18.04LTS	1	0	-	0	0
Desenvolupador	1	8	-	0	240
Preparació de l'entorn	**20 hores**				**166.2**
Dell Latitude 7490	1	1830	4	0.21	4.2
Dell Professional P2217H	1	250	4	0.03	0.6
Perifèrics	1	50	4	0.07	1.4
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	160
Mercurium	1	0	-	0	0
Nanos6	1	0	-	0	0
Integració d'SPDK	**200 hores**				**1662**
Dell Latitude 7490	1	1830	4	0.21	42
Dell Professional P2217H	1	250	4	0.03	6
Perifèrics	1	50	4	0.07	14
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	1600
Mercurium	1	0	-	0	0
Nanos6	1	0	-	0	0
GDB	1	0	-	0	0
GitLab	1	0	-	0	0
Avaluació de rendiment	**100 hores**				**831**
Dell Latitude 7490	1	1830	4	0.21	21
Dell Professional P2217H	1	250	4	0.03	3
Perifèrics	1	50	4	0.07	7
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	800
Mercurium	1	0	-	0	0
Nanos6	1	0	-	0	0
GDB	1	0	-	0	0
GitLab	1	0	-	0	0
Overleaf	1	0	-	0	0
Redacció de la memòria final	**60 hores**				**498.6**
Dell Latitude 7490	1	1830	4	0.21	12.6

Dell Professional P2217H	1	250	4	0.03	1.8
Perifèrics	1	50	4	0.07	4.2
Ubuntu 18.04LTS	1	0	-	0	0
VsCode	1	0	-	0	0
Desenvolupador	1	8	-	-	480
GitLab	1	0	-	0	0
Overleaf	1	0	-	0	0
Costos indirectes					**2300**
Despeses d'oficina	5	60	-	0	300
Salaris indirectes	100	20	-	0	2000
Total acumulat					**6828.95**
Contingència 15%					**1024.34**
Total sense IVA					**7853.29**
Total amb IVA 21%					**9502.48**

Taula 3: Pressupost detallat del projecte

Per tal d'elaborar la taula anterior s'ha tingut en compte el hardware i software descrit en la planificació, el seu cost, i el salari del desenvolupador. Com que el clúster de computació que finalment s'usarà per executar les proves finals encara no està concretat, no podem establir-ne el cost.

Per els costos indirectes s'ha tingut en compte els salaris de la resta d'actors del projecte, com el director o el ponent, i s'ha estimat la seva contribució en unes 100 hores. El concepte de "despeses d'oficina"agrupa el cost del despatx al K2M, l'electricitat, Internet, aigua, etc.

Per tal de cobrir possibles retrassos en la planificació, s'ha assignat una contingència del 15% destinada a cobrir aquests costos extra.

6.1.2 Control de Gestió

Tot i que s'ha intentat afinar al màxim la planificació temporal del projecte, és probable que algun obstacle en el desenvolupament n'allargui el temps, i com a conseqüència, augmenti el pressupost. Per a controlar aquestes situacions s'establiran reunions periòdiques amb el director del projecte.

Com s'ha explicat anteriorment, s'ha destinat una contingència del 15% del pressupost total per a poder cobrir augments de temps no previstos.

6.2 Desviacions respecte a la planificació final

En aquesta secció es detallen els canvis que han sorgit al projecte respecte a la planificació realitzada originalment, un cop ha estat enllestit.

6.2.1 Desviacions temporals

A la taula 4 apareix, desglossat per tasques, les diferències entre les dates de finalització previstes i les dates de finalització reals. Com es pot observar, hi ha hagut un retràs significatiu en el projecte que ha evitat que es pogués acabar d'acord a la planificació. Les causes del retràs són les següents:

- Va haver-hi problemes a l'hora de configurar l'entorn de desenvolupament amb totes les eines d'OmpSs-2 degut a una particularitat de la màquina on es treballava.
- Durant una de les fases de desenvolupament de programes de prova va existir un error de programació que va resultar molt difícil de depurar, i va comportar un retràs important en la resta de la planificació.
- Durant les proves de rendiment van sorgir petits errors d'implementació que no havien aparegut fins que no es va utilitzar proves amb paràmetres d'entrada significatius.

Tot i això, gràcies a que durant la etapa de planificació es va reservar suficient marge per a possibles imprevistos, s'ha pogut completar a temps.

6.2.2 Desviacions en el Pressupost

Com que es va reservar una contingència al pressupost que permetés cobrir despeses fruit de possibles retrassos, el projecte ha entrat dins del pressupost establert. També s'ha fet ús de les hores de còmput del clúster que no s'havia detallat a la planificació per encara no estar definit, però es fa difícil fer una estimació del cost que ha comportat aquestes hores.

Tasca	**Descripció**	**Observacions**	**Data Final Esperada**	**Data Final Real**
T1	Gestió de Projectes	Completat amb èxit (sense imprevistos)	29-03-2019	29-03-2019
T2	Estudi del *Programming Model i del runtime*	Completat amb èxit (sense imprevistos)	04-04-2019	04-04-2019
T3	Estudi d'SPDK	Completat amb èxit (sense imprevistos)	04-04-2019	04-04-2019
T4	Plantejament general del projecte	Completat amb èxit (sense imprevistos)	09-04-2019	09-04-2019
T5	Preparació de l'entorn de desenvolupament	Alguna complicació per instal·lar el *runtime*	12-04-2019	14-04-2019
T6	Integració d'SPDK amb Nanos6 i OmpSs-2	Complicacions importants degut a un problema difícil de depurar en la implementació de la llibreria	17-05-2019	25-05-2019
T7	Programes de prova i avaluació de rendiment	Alguns retrassos per problemes d'implementació que s'han detectat a l'hora d'utilitzar casos d'ús reals	04-06-2019	14-06-2019
T8	Redacció de la memòria final	Completat amb èxit un cop els resultats han estat llestos	14-06-2019	24-06-2019

Taula 4: *Diferències entre les dates de finalització previstes i les dates de finalització finals*

7 Sostenibilitat i compromís social

En aquesta secció es reflexionarà i s'analitzarà la sostenibilitat del projecte en tres àmbits diferents: Econòmic, Ambiental i Social.

7.1 Dimensió Econòmica

- **Has quantificat el cost (recursos humans i materials) de la realització del projecte? Quines decisions has pres per reduir el seu cost? Has quantificat aquest estalvi?**
  S'ha tingut en compte tots els recursos utilitzats per a cada tasca, excepte el clúster computacional perquè faltava per concretar a l'hora de fer la planificació. A més, s'han tingut en compte despeses indirectes. Tot el software usat és lliure o gratuït, així que s'ha minimitzat força el cost en aquest aspecte

- **S'ha ajustat el cost previst inicialment al final? Has justificat les diferències?**
  Com que s'ha hagut d'utilitzar la contingència per culpa d'un retràs en la implementació, al final s'ha utilitzat tot el pressupost que es va destinar al projecte.

- **Quin cost estimes que tindrà el projecte durant la seva vida útil? Es podria reduir aquest cost per fer-lo més viable?**
  Aquest projecte no té un cost associat a la seva vida útil, ja que no s'ha de mantenir cap servei ni pagar cap cost addicional un cop s'ha acabat.

- **S'ha tingut en compte el cost dels ajustos/actualitzacions/reparacions durant la vida útil del projecte?**
  Cap la possibilitat d'estendre la llibreria per suportar més característiques en un futur, però això s'emmarcaria dins d'un altre projecte i caldria realitzar un nou anàlisi econòmic.

- **Podrien produir-se escenaris que perjudiquessin la viabilitat del projecte?**
  La llibreria serà publicada amb una llicència permissiva, per tant no es busca obtenir rendibilitat econòmica.

7.2 Dimensió Ambiental

- **Has quantificat l'impacte ambiental de la realització del projecte? Quines mesures has pres per reduir-ne l'impacte? Has quantificat aquesta reducció?**
  Durant la planificació del projecte no s'havia decidit encara quina configuració s'utilitzaria per a realitzar les proves de rendiment finals, així que el càlcul del consum elèctric que ha comportat no s'ha pogut incloure. Els recursos materials utilitzats durant tot el projecte seran reutilitzats en un futur, com ara el clúster de còmput o l'ordinador personal que ha cedit el BSC-CNS.

- **Si fessis de nou el projecte, podries realitzar-lo amb menys recursos?**
  Sí, ja que amb els coneixements que he adquirit durant la realització del projecte s'haguessin pogut estalviar hores de computació, que involucren una despesa elèctrica important, i d'hores de treball.

- **Quins recursos estimes que es faran servir durant la vida útil del projecte?**

Quin serà l'impacte ambiental d'aquests recursos?
Durant la vida útil del projecte servirà per a ser executat en entorns d'HPC, que comporten una despesa d'electricitat. La majoria d'electricitat generada avui dia s'obté per la crema de combustibles fòssils, que tenen un impacte medioambiental important.

- **El projecte permetrà reduir l'ús d'altres recursos? Globalment, el projecte millorarà o empitjorarà la petjada ecològica?**
  Globalment ajudarà a reduir l'ús elèctric ja que comportarà una reducció dels temps d'execució dels programes on s'inclogui.

- **Podrien produir-se escenaris que fessin augmentar la petjada ecològica del projecte?**
  No, ja que no genera cap residu ni generarà un impacte addicional al tractar-se de *software*-

7.3 Dimensió Social

- **La realització d'aquest projecte ha implicat reflexions significatives en l'àmbit personal, professional o ètic de les persones que hi han intervingut?**
  Ha aportat un important aprenentatge sobre la organització del temps i de la feina. A més, ha implicat una millora del nivell d'anglès i millorat la meva capacitat de colaborar amb altres departaments o treballadors, així com de guanyar importants coneixements professionals que sens dubte seran útils en un futur.

- **Qui es beneficiarà de l'ús del projecte? Hi ha algun col·lectiu que pugui veure's perjudicat pel projecte? En quina mesura?**
  La millora de rendiment prevista que pot proporcionar aquest projecte està enfocat a l'ús en càlculs d'HPC científics, i una millora en el seu rendiment implicarà una possible millora en el número de descobriments científics, que poden aportar una millor qualitat de vida a la societat. No es preveu que pugui perjudicar a cap col·lectiu.

- **En quina mesura soluciona el projecte el problema plantejat inicialment?**
  S'ha aconseguit una integració eficient i funcional d'SPDK dins del programming model d'OmpSs-2, així que en aquest sentit, s'ha aconseguit els resultats que s'esperaven.

- **Podrien produir-se escenaris que fessin que el projecte fos perjudicial per a algun segment particular de la població?**
  La llibreria està enfocada a l'àmbit científic, així que generalment, a excepció de que s'utilitzés en una aplicació militar, cosa poc probable, no es preveu que sigui perjudicial per a cap col·lectiu, ans al contrari.

- **Podria crear el projecte algun tipus de dependència que deixés als usuaris en posició de debilitat?**
  Tot el material generat a través d'aquest projecte serà alliberat per al seu ús públic, per tant, qualsevol usuari podria referir-se a ell en cas de tenir dubtes o voler arreglar errors, i no es quedarien amb un producte abandonat.

8 *Task Aware* SPDK

En aquesta secció es detallarà la llibreria implementada en aquest projecte que s'encarrega d'integrar SPDK i OmpSs-2, anomenada *Task Aware* SPDK (TASPDK).

8.1 Objectiu

L'objectiu principal de TASPDK és oferir una eina eficient que estigui integrada dins del *programming* model OmpSs-2 per tal d'expressar dependències entre tasques que treballin amb operacions d'E/S, a través de l'ús d'SPDK per explotar els beneficis de rendiment i el paral·lelisme dels dispositius NVMe. La llibreria oferirà mecanismes que segueixin la especificació d'OmpSs-2 per expressar dependències de dades entre tasques que efectuïn accessos a fitxers TASPDK. Per fer-ho, la llibreria proporcionarà al programador un espai virtual de memòria que es farà servir per expressar el domini de dependències de l'espai de dades del dispositiu. A la figura 9 es mostra un breu exemple de com aconseguir aquesta sincronització utilitzant les crides de TASPDK i el *programming model* OmpSs-2. En aquest exemple, es creen dues tasques, on cada una opera amb la meitat d'un fitxer, i una tercera exporta el resultat dels càlculs. Per tal de sincronitzar les dues tasques que creen les dades amb la tercera, que les consumirà, és necessari expressar d'alguna forma les dependències que presenten amb el fitxer, cosa que es fa a través del punter virtual retornat per *taspdk_ addr*.

```
size_t FILE_SIZE = 4096;
taspdk_fd fd = taspdk_fd taspdk_open("Example", FILE_SIZE, ...);
void *file_addr = taspdk_addr(fd);

#pragma oss task inout(file_addr[0;FILE_SIZE/2]
do_computation(...)

#pragma oss task inout(file_addr[FILE_SIZE/2;FILE_SIZE/2]
do_computation(...)

#pragma oss task in(file_addr[0;FILE_SIZE])
export_results(..)
```

Figura 9: *Exemple d'ús del punter virtual*

A part, TASPDK s'encarregarà d'efectuar les operacions d'inicialització d'SPDK, com ara registrar el controlador del dispositiu NVMe que es vol utilitzar, gestionar les cues NVMe i oferir una interfície similar a les crides POSIX[21] de manipulació de fitxers, tant de forma síncrona com asíncrona, que resultarà familiar a la majoria de programadors.

A la figura 10 es mostra el codi necessari per tal de llegir un bloc de dades d'un dispositiu utilitzant SPDK, manipular-lo, i tornar-lo a escriure i, a la figura 11, el codi necessari per fer aquesta operació si s'utilitza TASPDK. Com es pot comprovar, la interfície és molt més còmoda d'utilitzar i encapsula el funcionament intern d'SPDK, com ara el l'ús de *callbacks* o de *polling* per tal de comprovar la finalització de les peticions. A més, aplicacions existents podrien passar a utilitzar TASPDK fent canvis mínims en el seu codi, simplement canviant les crides de manipulació de fitxers POSIX *pread* o *pwrite* per *taspdk_ read* o *taspdk_ write*. Fer aquest canvi utilitzant SPDK directament implicaria una reestructuració del codi més substancial.

```c
// Callback que cridarà SPDK quan es completi la operació
void io_callback(void *cb_data){
    int *request_completed = *((int *)cb_data);
    request_completed = 1;
}
/* S'ha omès el codi necessari per registrar SPDK amb el dispositiu NVMe
* per brevetat */
int main(int argc, char **argv){
    // S'obté la mida d'un sector del dispositiu ja que les peticions a SPDK es fan
        a nivell de bloc.
    size_t block_size = spdk_nvme_ns_get_sector_size(ns);
    size_t buffer_size = block_size;
    size_t num_blocks = buffer_size / block_size;
    size_t offset = 0;
    // Es reserva memòria especial per fer operacions DMA
    void *buf = spdk_malloc(buffer_size, 0, NULL, SPDK_ENV_SOCKET_ID_ANY,
        SPDK_MALLOC_DMA);

    int request_completed = 0;
    // Es crea una cua NVMe a la qual es farà la petició
    struct spdk_nvme_qpair *qpair = spdk_nvme_ctrlr_alloc_io_qpair(env->ctrlr, NULL
        , 0);

    // Llegir un bloc
    spdk_nvme_ns_cmd_read(ns, qpair, buf, offset, num_blocks, io_callback, (void *)
        &request_completed, 0);
    // Polling per assegurar que la operació ha acabat
    while(!request_completed);

    make_computation(buf, buffer_size);

    request_completed = 0;
    // Escriure un bloc
    spdk_nvme_ns_cmd_write(ns, qpair, buf, offset, num_blocks, io_callback, (void
        *)&request_completed, 0);
    // Polling per assegurar que la operació ha acabat
    while(!request_completed);

    spdk_free(buf);
    spdk_nvme_ctrlr_free_io_qpair(qpair);
    /* S'ha omès el codi necessari per alliberar el dispositiu d'SPDK per brevetat
        */
    return 0;
}
```

Figura 10: Codi necessari per llegir un bloc de dades utilitzant SPDK

```
int main(int argc, char **argv){
    // Inicialització de TASPDK
    taspdk_init();
    // S'obté la mida d'un sector del dispositiu ja que les peticions a SPDK es fan
        a nivell de bloc.
    size_t block_size = taspdk_get_block_size();
    size_t buffer_size = block_size;
    size_t num_blocks = buffer_size / block_size;
    size_t offset = 0;
    // Es reserva memòria especial per fer operacions DMA
    void *buf = taspdk_malloc(buffer_size);

    // Es crea un fitxer TASPDK
    int fd = taspdk_open("Example", buffer_size, TASPDK_O_RDWR | TASPDK_O_CREAT);
    // Efectuar la petició al disc
    taspdk_read(fd, buf, offset, buffer_size);

    make_computation(buf, buffer_size);

    taspdk_write(fd, buf, offset, buffer_size);
    taspdk_free(buf);
    taspdk_cleanup();
    return 0;
}
```

Figura 11: Codi necessari per llegir un bloc de dades utilitzant SPDK

8.2 Requisits

Aquesta llibreria requereix una versió del *runtime* Nanos6 que implementi les APIs de *pause/resume* i d'events externs i *polling services*, explicades a continuació. Aquestes APIs formen part d'OmpSs-2 i de Nanos6 des de la *release* 18.06 d'OmpSs-2 [22].

També és necessària una versió d'SPDK 18.04 o més recent.

8.2.1 *Pause/Resume* API

A la figura 12 apareixen les crides neccesàries de les quals ha de disposar la implementació d'OmpSs-2 utilitzada les operacions d'E/S síncrones.

La primera crida retorna un punter amb dades que el *runtime* necessitarà per pausar la tasca més endavant. La segona i la tercera, pausen la tasca i en continuen l'execució, respectivament. Mentre una tasca està pausada, aquell *thread* està lliure per executar qualsevol altra tasca que estigui disponible.

Aquesta API s'utilitza en la llibreria per tal de bloquejar la tasca que realitza una operació d'E/S fins que aquesta s'ha completat.

```
// Returns pointer to runtime data
void *nanos6_get_current_blocking_context();

// Blocks and unblocks tasks
void nanos6_block_current_task(void *blocking_context);
void nanos6_unblock_task(void *blocking_context);
```

Figura 12: *Capçaleres de les crides de la API Pause/Resume necessàries*

```
// Returns an event counter
void *nanos6_get_current_event_counter();

// Increases and decreases the event counter
void nanos6_increase_current_task_event_counter(void *event_counter, unsigned int
    increment);
void nanos6_decrease_task_event_counter(void *event_counter, unsigned int
    decrement);
```

Figura 13: *Capçaleres de les crides de la API d'events externs necessàries*

8.2.2 API d'Events Externs

A la figura 14 apareixen les crides neccesàries de les quals ha de disposar la implementació d'OmpSs-2 utilitzada per tal de soportar les operacions d'E/S asíncrones.

La primera crida retorna un punter amb dades que el *runtime* necessitarà per tal de comptar els events externs que té registrada aquesta tasca. La segona i la tercera augmenten i redueixen, respectivament, aquest comptador de forma atòmica.

Aquesta API permet deslligar la finalització d'una tasca de la alliberació de les seves dependències. Així, una tasca pot acabar, i alliberar els recursos que utilitza, pero només alliberarà les seves dependències quan el seu comptador d'events externs és zero. Aquesta API és necessària per tal de realitzar operacions d'E/S asíncrones.

8.2.3 *Polling Services* API

A la figura 14 apareixen les crides neccesàries de les quals ha de disposar la implementació d'OmpSs-2 utilitzada per tal de soportar el *polling* de les operacions d'E/S que s'han registrat.

La primera crida registra i desregistra, respectivament, una funció que s'executarà periòdicament per part del *runtime*.

Aquesta API s'utilitzarà per tal de fer un *polling* periòdic de l'estat de les cues NVMe creades per SPDK per tal de confirmar quines operacions d'E/S han acabat. SPDK ofereix la possibilitat de passar un *callback* que s'executarà quan es faci *polling* d'una petició que ja ha acabat. Aquest mecanisme de *callback* és el que s'usarà per tal d'implementar la continuació de les tasques o el decrement del contador d'events externs.

```
/* The nanos6_polling_service_t signature is the following:
   int (*nanos6_polling_service_t)(void *data);
*/
  void nanos6_register_polling_service(char const *name, nanos6_polling_service_t
      service, void *data);
  void nanos6_unregister_polling_service(char const *name,
      nanos6_polling_service_t service, void *data);
```

Figura 14: *Capçaleres de les crides de la API de Polling Services necessaries*

8.3 Interfície de TASPDK

La interfície pública de la nova llibreria TASPDK es troba continguda en un fitxer *taspdk.h*. Com s'ha comentat anteriorment, la interfície de llibreria ha estat dissenyada per oferir una API similar a l'API POSIX per realitzar operacions amb fitxers. Totes les crides començen amb el prefix *taspdk_* seguit de la operació que es vol realitzar, com ara *taspdk_ open()* o *taspdk_ read()*. A continuació es comentaran les parts més rellevants.

8.3.1 Inicialització

```
// Initializes and cleans up TASPDK data
int taspdk_init(void);
void taspdk_cleanup(void);
```

Figura 15: *Capçaleres de les crides d'inicialització i finalització de TASPDK*

Tal com s'ha comentat a la secció SPDK de l'estat de l'art, abans d'inicialitzar TASPDK s'haurà d'haver executat l'script de configuració que proporciona SPDK.

A la figura 15 apareixen les dues crides que inicialitzen i finalitzen, respectivament, la llibreria TASPDK.

Primer es fa un reconeixement de quins dispositius NVMe existeixen en el sistema, i s'intenta registrar el primer que es troba. Actualment TASPDK no soporta utilitzar múltiples dispositius, i es fa servir el primer disponible.

Un cop s'ha registrat el dispositiu correctament s'analitza de quants *cores* disposa el *runtime*, i quin és el número màxim de cues d'E/S que soporta el dispositiu registrat. A continuació es creen n cues on n és el mínim entre el número de *cores* i el número màxim de cues. Cada processador queda associat a una cua de la forma $queue\_number = core\_number \bmod number\_of\_queues$.

Com que el fet de registrar una petició en una cua no és *thread-safe*, també es creen n locks. Quan TASPDK efectuï una petició d'E/S sobre una cua primer es efectuarà un *spinlock* sobre el lock assignat a aquella cua per assegurar una correcta sincronització.

A continuació es registra amb nanos6 el *polling service* que s'encarrega de comprovar quines peticions s'han completat.

Finalment, s'inicialitzen les estructures de dades internes de TASPDK necessàries per gestionar els diferents fitxers que es creïn en l'aplicació.

La funció *taspdk_ cleanup* desregistra el *polling service*, comprova que hagin acabat totes les peticions, desregistra el dispositiu i allibera les cues i altres estructures internes reservades durant la inicialització.

8.3.2 Creació de fitxers

```
// Opens a file and gets the virtual address where it's mapped
taspdk_fd taspdk_open(char *name, size_t size, int flags);
void *taspdk_addr(taspdk_fd fd);
```

Figura 16: *Capçaleres de les crides TASPDK per crear fitxers*

A la figura 16 apareixen dues crides lligades a la creació de fitxers.

La *taspdk_ open* obre un fitxer existent de nom *name* i en retorna l'identificador. Flags és un o una combinació dels següents flags: TASPDK_O_RDONLY, TASPDK_O_WRONLY, TASPDK_O_RDWR o TASPDK_O_CREAT. Si es passa el flag TASPDK_O_CREAT, es crea un fitxer de mida *size*. SPDK exposa els dispositius NVMe com a dispositius de blocs, per tant TASPDK reserva un rang de blocs en funció de la mida del fitxer i els assigna a un identificador de fitxer. Dos fitxers no compartiran blocs. A la figura 17 es mostra la implementació del procés que se segueix per implementar la creació de nous fitxers

```
  // Creates new TASPDK file entry and returns the file descriptor
int taspdk_new_fd_entry(char *name, size_t size, int flags){
    //Increases the file descriptor number
    unsigned int fd = atomic_fetch_add(&next_fd,1);
    assert(fd < FD_MAX_NUMBER);

    taspdk_fd_table[fd].name = name;
    /* Increases the global virtual pointer used to perform dependency checking
    * and associates the file to the beggining of the mapping reserved for it
    */
    taspdk_fd_table[fd].ptr = __taspdk_mmap(fd, size);

    /* Reserves the blocks that will be used for the file according to the
    * size specified by the user.
    */
    __taspdk_alloc(fd, size);
    taspdk_fd_table[fd].size = size;
    taspdk_fd_table[fd].permissions = flags;
    return fd;
}
```

Figura 17: *Procés de creació d'un fitxer TASPDK*

La crida *taspdk_ addr* retorna un punter a una direcció de memòria a partir de la qual es considera que està mapejat el fitxer. Aquest mapeig no és real, si no que es tracta d'un rang de direccions que s'usaran com a sentinella. Accedir a aquesta memòria suposarà, doncs, una excepció per accés invàlid.

Aquest mecanisme és important per tal de poder expressar dependències de tasques d'OmpSs-2 sobre fitxers de TASPDK. Les dependències de dades es garanteixen especificant en les directives de les tasques rangs de direccions de memòria amb la qual operaran, tant com de lectura

```
// Synchronous API
int taspdk_read(taspdk_fd fd, void *buf, size_t offset, size_t count);
int taspdk_write(taspdk_fd fd, void *buf, size_t offset, size_t count);

// Asynchronous API
int taspdk_aread(taspdk_fd fd, void *buf, size_t offset, size_t count);
int taspdk_awrite(taspdk_fd fd, void *buf, size_t offset, size_t count);

// DMA Memory allocation
void *taspdk_malloc(size_t size);
void taspdk_free(void *buf);

size_t taspdk_get_block_size(taspdk_fd fd);
```

Figura 18: *Capçaleres de les crides per fer lectures, escriptures i reserva de memòria*

com d'escriptura. Per tal de garantir la sincronització entre tasques que realitzaran operacions amb el disc, doncs, és necessari expressar aquestes dependències com a accesos a una memòria virtual que representa el conjunt de dades del fitxer. Dues tasques que necessitin sincronitzar accesos al disc poden fer servir el punter retornat per *taspdk_addr* per aconseguir-ho.

8.3.3 Lectures i Escritures

A la figura 18 apareixen les crides relacionades amb les operacions de lectures i escritures, així com de reserva de memòria. Per tal que el driver NVMe pugui realitzar la operació DMA amb el buffer que subministra l'usuari, és necessari que aquesta memòria sigui de tipus *pinned*, és a dir, que la seva direcció física no canviï. Un buffer que estigui reservat fent servir una crida *malloc* estàndar no té aquesta garantia, així que és important fer servir *taspdk_malloc* per reservar els buffers que es faran servir a l'hora d'operar amb el disc. També és important no barrejar les crides *malloc* i *free* de TASPDK amb les de la llibreria estàndar; utilitzar *taspdk_free* per alliberar memòria reservada amb *malloc* resulta en un error, i viceversa.

Les crides *taspdk_read* i *taspdk_write* generen peticions síncrones, és a dir, que bloquegen l'execució de la tasca on es troben fins que s'han completat, de lectura i escriptura, respectivament, sobre el fitxer especificat al paràmetre *fd*. El paràmetre *buf* és un punter a una zona de memòria *pinned* d'on es llegiran els continguts, en el cas del *write*, o on s'escriuran, en cas del read. Els paràmetres *offset* i *count* indiquen, en bytes, l'*offset* dins del fitxer a partir d'on es realitzarà l'operació i la mida. Aquesta mida ha de ser múltiple de la mida de bloc que utilitzi el dispositiu on s'emmagatzema el fitxer; per tal de saber quina mida és es pot utilitzar la crida *taspdk_get_block_size*.

Les crides *taspdk_aread* i *taspdk_awrite* generen peticions asíncrones i es comporten de forma anàloga a les crides síncrones, exceptuant que no bloquegen la tasca, sino que utilitzen la API d'events externs explicats a la secció *Capçaleres de les crides de la API de Polling Services necessaries*. Aquestes crides, doncs, no bloquegen la tasca fins que l'operació d'E/S es completa, si no que permeten que se segueixi executant. Això presenta alguna implicació en tasques aniuades que utilitzin les crides asíncrones. En el cas de la figura 19 podem veure com el resultat que s'obtindrà és indefinit, ja que al trobar-se la clausula final, les tasques que es creïn seran executades de forma sèrie i la tercera tasca no té garantitzada que els resultats en els buffers estiguin llestos.

```
int condition = true;

#pragma oss task final(condition==true)
{
    /* Les tasques creades aquí dins s'executaran de
    forma sèrie ja que la clàusula final evalua a true */
    void *a = taspdk_malloc(...);
    void *b = taspdk_malloc(...);

    #pragma oss task out(a[...])
    taspdk_aread(fd, a, ...);

    /* ERROR! No hi ha cap garantia que a tingui
    el contingut correcte */
    #pragma oss task in(a[...]) out(b[...])
    do_computation(a,b);
}

#pragma oss taskwait
```

Figura 19: *Exemple de la interacció de la clàusula final i les crides asíncrones*

9 Resultats i Avaluació

Per tal d'evaluar el rendiment de TASPDK s'han elaborat una sèrie d'aplicacions intensives en E/S, utilitzant un patró comú: es simula disposar d'un *dataset* massa gran per cabre a la memòria principal, i els algoritmes es duen a terme per blocs, accedint a les dades en fragments que es puguin processar. Aquest tipus d'algoritmes s'anomenen *out-of-core*[23].

S'han implementat versions de l'algoritme de *clustering* K-means, *External* MergeSort i un filtre d'imatge per convolució.

9.1 Metodologia

Totes les mostres de rendiment s'han executat en un node de computació format per:

- Dos processadors Intel Xeon E5-2690v4 a 2.60GHz, que disposa de 14 nuclis i 2 fils d'execució per nucli. En total, el node disposa de 28 nuclis i 56 fils d'execució.
- Un disc dur Intel Optane 905P de 960GB. Aquest disc dur actualment és l'estat de l'art en dispositius NVMe, i ofereix un ample de banda de lectura i escriptura seqüencials de fins a 2,700 i 2,200 MB/s, respectivament.
- 126GB totals de memòria RAM DDR4

Per prendre les mesures s'han realitzat les proves diverses vegades i s'ha calculat la mitjana aritmètica. Les mesures temporals s'han pres a través de la instrumentació del codi font.

9.2 *Overhead* respecte a SPDK

Abans de comentar els resultats obtinguts als *benchmarks* obtinguts, seria interessant comprovar quin *overhead* està introduïnt TASPDK respecte a SPDK i si s'ha conseguit l'objectiu que aquest fos mínim. Per tal de fer aquesta prova s'ha fet un programa que simplement omple o llegeix un número elevat de blocs, sense fer cap càlcul amb les dades, i s'ha mesurat el temps requerit. Una versió utilitzava les crides de la llibreria TASPDK per interactuar amb el disc, i l'altra operava amb SPDK directament.

Tal com es pot comprovar a la figura 20, el bandwidth assolit quadra amb el que ens publicita el fabricant, i, tot i que el rendiment utilitzant SPDK és lleugerament superior, la pèrdua de rendiment introduïda per TASPDK és mínima.

9.3 *External* MergeSort

L'algoritme de mergesort tradicional treballa amb un array de dades que cap a la memòria principal. Si es vol aplicar a un *dataset* de mida més gran, una opció és utilitzar l'*External* MergeSort. L'algoritme, que és del tipus *divide and conquer*, consisteix en el següent:

1. Es llegeixen n blocs, anomenats *runs*, de mida BS d'un fitxer de mida N, on BS és una mida suficientment petita per caber en memòria.

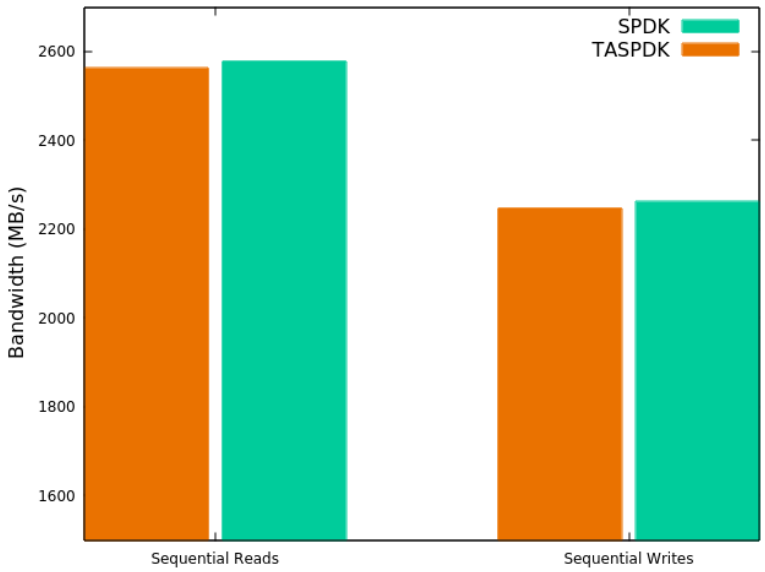


Figura 20: Comparació de l'ample de banda màxim assolible entre SPDK i TASPDK

2. S'ordena cada un dels n blocs, utilitzant algoritmes *in-memory* convencionals, com per exemple, un *QuickSort*, i s'escriu un altre cop al disc.

3. Es divideix recursivament el fitxer en dues meitats, A i B; quan s'arriba a una fracció de problema de mida BS es procedeix a fer un *merge* d'A i B i s'escriu al disc. El *merge* funciona de la següent forma: s'utilitzen dos *buffers* d'entrada, a i b, i un de sortida c, de mida BS, i, per cada element d'a i b, s'escriu en c el que compleix la condició de la comparació. Quan a o b es buiden, es llegeixen noves dades d'A i B, respectivament, si en queden. Quan c s'omple, s'escriu al fitxer. Es segueix fins que s'han ordenat tots els elements d'A i de B.

4. Es pugen els nivells de recursivitat fins que tot el fitxer quedi ordenat, en total, $\log(n)$ operacions de *merge*.

Aquest algoritme presenta una fàcil paral·lelització a nivell de tasca. En la primera fase, la creació de *runs*, es pot descomposar en n tasques independents, on cada una llegeix, ordena i escriu un bloc. En la fase recursiva, cada una de les operacions de merge en un nivell de recursivitat es poden executar en paral·lel respecte les altres. A més, utilitzar operacions de disc asíncrones permet efectuar operacions de *merge* d'unes dades mentre s'estan llegint o escrivint d'altres.

A la figura 21 apareix el codi que genera les tasques recursives. Primer es comprova que s'hagi arribat a la mida de bloc mínima que ja s'haurà ordenat en memòria a la fase de creació de *runs*. Es genera una tasca per ordenar la part esquerra del fitxer i una per la part dreta. Les dependències es garanteixen dins de la operació de *merge*, tal com es pot veure a la figura 22.

Destacar de la operació merge la condició que s'utilitza per saber quan s'ha d'acabar de fer el merge dels blocs. Una opció seria comprovar al final de cada iteració si s'ha escrit el número de blocs necessaris mitjançant una clàusula *taskwait*. El problema d'aquest mètode és que aquesta sincronització és costosa i innecessària. El que s'ha optat per fer és dos bucles. Sabem que si s'ha de fer un merge de n blocs totals, com a mínim s'hauran d'efectuar n iteracions, escrivint a disc un bloc en cada una. En cada iteració però, pot ser que no s'hagi d'escriure al disc ja que hem

```
#pragma oss task weakinout(src_fd_ptr[start:end]) weakinout(tmp_fd_ptr[start:end])
    weakinout(a[0;BS],b[0;BS],c[0;BS]) label(merge)
void merge(taspdk_fd src_fd, taspdk_fd tmp_fd, void *a, void *b, void *c, unsigned
    long long start, unsigned long long end, void *src_fd_ptr, void *tmp_fd_ptr, bool
    is_final);

#pragma oss task weakin(src_fd_ptr[start:end]) weakinout(tmp_fd_ptr[start:end]) final(
    depth==DEPTH_LIMIT) label(mergesort_rec)
void merge_sort_rec(taspdk_fd src_fd, taspdk_fd tmp_fd, unsigned long long start,
    unsigned long long end,
                           void *src_fd_ptr, void *tmp_fd_ptr, int depth){
     unsigned long long mid = (start + end) / 2;

     if((end-start)+1 <= BS){
          return;
     }

     if(start < mid){
          merge_sort_rec(tmp_fd, src_fd, start, mid, tmp_fd_ptr, src_fd_ptr, depth+1)
              ;
     }

     if (mid+1 < end) {
          merge_sort_rec(tmp_fd, src_fd, mid + 1, end, tmp_fd_ptr, src_fd_ptr, depth
              +1);
     }
     void *A = taspdk_malloc(BS);
     void *B = taspdk_malloc(BS);
     void *C = taspdk_malloc(BS);

     merge(src_fd, tmp_fd, A, B, C, start, end, src_fd_ptr, tmp_fd_ptr);
     ...
}
```

Figura 21: Codi que demostra la creació de tasques recursives

d'aturar l'operació de merge per omplir algun dels buffers d'entrada. Així doncs, es realitzen n iteracions i després es sincronitzen les tasques. S'actualitza el número de blocs que queden per escriure restant el número de blocs escrits, m blocs escrits, de n blocs totals, i es torna a llançar $n - m$ iteracions. Se segueix així fins que s'han escrit tots els blocs necessaris. Aquesta és una de les limitacions d'OmpsS-2, és molt fàcil expressar dependències de dades entre tasques, però expressar dependències de control és més costós.

```
unsigned int blocks_to_flush = ((end-start)+1)/BS;
     while(blocks_to_flush > 0){
          int minimum_iterations = blocks_to_flush;
          for(int i=0; i<minimum_iterations; i++) {
               #pragma oss task in(a[0;BS], b[0;BS]) out(c[0;BS]) inout(index_a,
                   index_b, index_c) label(merge_blocks)
               merge_blocks((int *)a, &index_a, (int *)b, &index_b, (int *)c, &
                   index_c);

               #pragma oss task inout(off_a, index_a) out(a[0;BS]) in(src_fd_ptr[
                   off_a;BS]) firstprivate(mid, BS) \
               label(refill_a)
               {
                    if(index_a * sizeof(int) == BS && off_a < mid+1){
                         //refill buffer a
                         while(taspdk_read(src_fd, a, off_a, BS)<0);
                         off_a += BS;
                         index_a = 0;
                    }
               }
               #pragma oss task inout(off_b, index_b) out(b[0;BS]) in(src_fd_ptr[
                   off_b;BS]) firstprivate(end, BS) \
               label(refill_b)
               {
                    if(index_b * sizeof(int) == BS && off_b < end+1){
                         //refill buffer b
                         while(taspdk_read(src_fd, b, off_b, BS)<0);
                         off_b += BS;
                         index_b = 0;
                    }
               }
               #pragma oss task inout(index_c, off_c, blocks_to_flush) in(c[0;BS])
                   out(tmp_fd_ptr[off_c;BS]) firstprivate(BS) \
               label(flush_c)
               {
                    if(index_c*sizeof(int) == BS){
                         while(taspdk_write(tmp_fd, c, off_c, BS)<0);
                         blocks_to_flush--;
                         index_c = 0;
                         off_c += BS;
                    }
               }
          }
          #pragma oss taskwait in(blocks_to_flush)
     }
}
```

Figura 22: Codi que demostra la sincronització entre tasques dins de la operació de merge

Per les següents mesures s'ha utilitzat un fitxer d'entrada d'1GB i mides de bloc de fins a

256MB. Com que es reserven 3 blocs a cada operació de merge, aquesta és la mida màxima que satisfà el requisit d'operar amb menys menys memòria que la mida del fitxer.

9.3.1 *Strong Scalability*

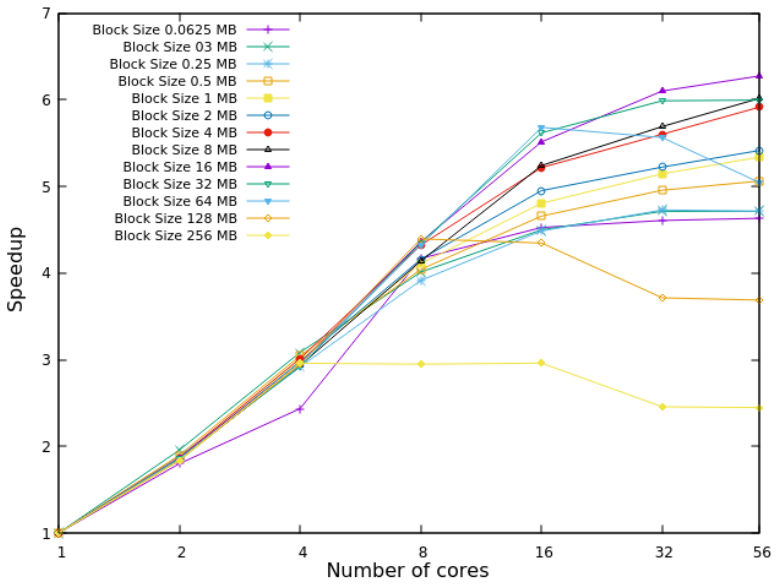


Figura 23: Strong scaling de l'*External* MergeSort amb un fitxer d'1GB per diverses mides de bloc

A la figura 23 es mesura la *strong scalability*, és a dir, la millora de rendiment que presenta el programa, donada una mida de problema fixa, respecte a la versió seqüencial quan s'augmenta el número de processadors utilitzats, per a cada mida de bloc que s'ha mesurat. Tal com es pot comprovar, la implementació no escala de forma lineal, això és degut a diversos aspectes: Primer, el paral·lelisme potencial que presenta l'algoritme cau a mesura que es puja en nivells de recursivitat; els nivells superiors necessiten que s'hagi completat els nivells inferiors per continuar. A més, la operació de merge és per naturalesa seqüencial, ja que s'han de llegir els fitxers en ordre per tal d'assegurar un resultat correcte. A mesura que es puja en l'arbre de recursivitat les operacions de *merge* es fan més costoses, ja que augmenta el número de blocs, així que a més, s'està creant un desbalanceig de càrrega. Es pot observar com, a partir de 4 *cores*, la mida de bloc més gran, 256MB, no ofereix cap millora de rendiment amb més processadors. Això és degut a que l'arbre de recursivitat només tindrà $log_2(1GB/256MB) = log_2(4) = 2$ nivells de recursivitat. Amb 2 nivells de recursivitat s'efectuaran 4 crides al *in-memory sorting* que es poden executar en paral·lel, corresponents als nodes fulla. Aquest és el màxim nivell de paral·lelisme que es pot aconseguir, ja que les següents crides de *merge* només utilitzaran dos i un processador, respectivament, ja que la operació de *merge* és, per naturalesa, seqüencial. Això explica que, a partir de 4 processadors, amb la mida de bloc de 256MB no es guanyi rendiment.

Per altra banda, si la mida de bloc és massa petita, l'arbre de recursivitat serà més profund, cosa que permet explotar més paral·lelisme, però les crides de merge dels nivells superiors hauran d'efectuar moltes peticions d'E/S al disc, cosa que introduïrà un *overhead* que limitarà la escalabilitat. Per la mida de problema que hem tractat, la mida de bloc que equilibra millor la profunditat de l'arbre de recursivitat amb el menor número possible de operacions d'E/S i que, per tant, mostra un millor rendiment, és 16MB.

La figura 24 mostra l'eficiència en relació al número de *cores* utilitzats, com a mesura per

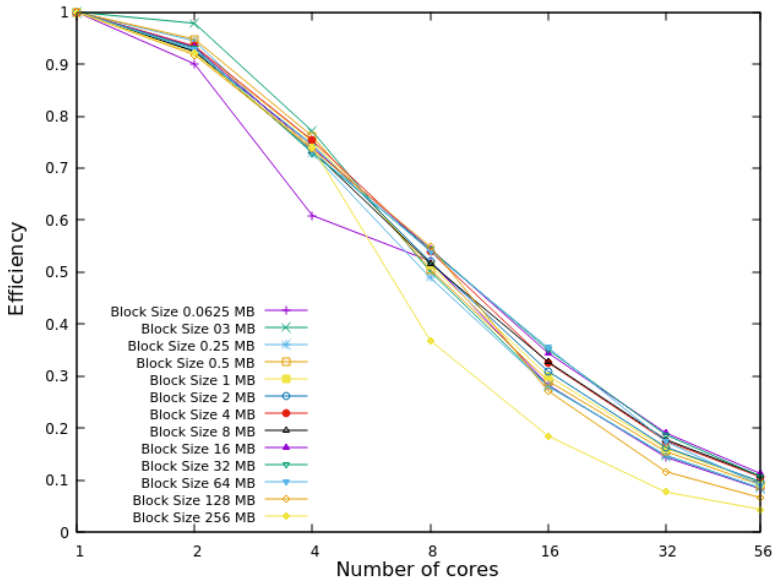


Figura 24: Eficiència de l'*External* MergeSort amb un fitxer d'1GB per diverses mides de bloc

complementar l'*speedup*. Com es pot veure, la mida de bloc que presenta una eficiència menor és 256MB, cosa que quadra amb el que s'ha vist a la figura de la *Strong Scalability*

S'ha omès els gràfics que utilitzaven les crides asíncrones perquè no presentaven cap diferència de rendiment destacable.

9.3.2 *Bandwidth*

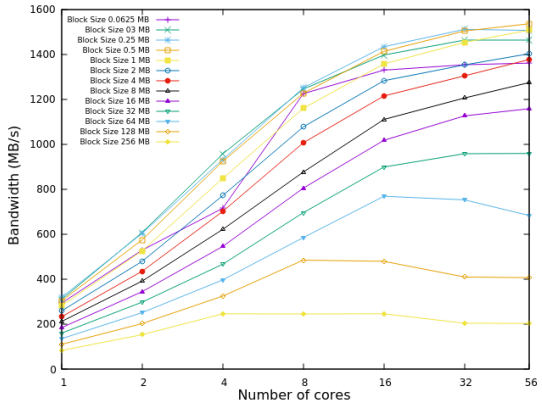
Per tal de calcular el *bandwidth* amb el disc s'ha de tenir en compte que el número de bytes transferits amb el dispositiu varia, donada una mida de fitxer, en funció de la mida de bloc escollida. Així, amb un fitxer de mida F bytes, una mida de bloc BS bytes un número de blocs n tal que $F = BS * n$ i un temps d'execució t, primer s'efectuen n lectures i n escriptures per tal de fer la creació dels *runs*. Després, s'efectuen $\log_2(n)$ crides recursives a *merge*, on a cada nivell es llegeixen n blocs i s'escriuen n blocs. En total, doncs el *bandwidth* resultant vindrà donat per la equació següent:

$$Bandwidth = BS * \frac{2n(1 + \log_2(n))}{t}$$

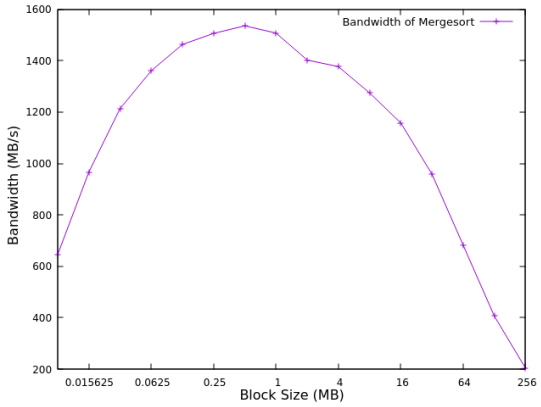
A la figura 25 a) es pot observar com per un fitxer d'1GB, la mida de bloc òptima per obtenir el *bandwidth* màxim se situa en els 512KB. Si representem la gràfica del *bandwidth* respecte a la mida de bloc, utilitzant 56 *cores*, a la figura 25 b) podem veure la característica forma d'U invertida que sol aparèixer en aquest tipus de proves.

Cal destacar que el mínim temps d'execució no s'obté maximitzant el *bandwidth*, sinó amb una mida de bloc de 16MB, tal com es pot comprovar a la figura 26 b). Tal com s'ha comentat abans, això ve del fet que interessa trobar una mida de bloc que permeti suficient paral·lelisme, però que involucri el mínim d'operacions d'E/S.

Altra vegada podem observar que no existeix una diferència substancial de *bandwidth* entre la versió síncrona i la asíncrona, així que s'han omès les gràfiques corresponents.

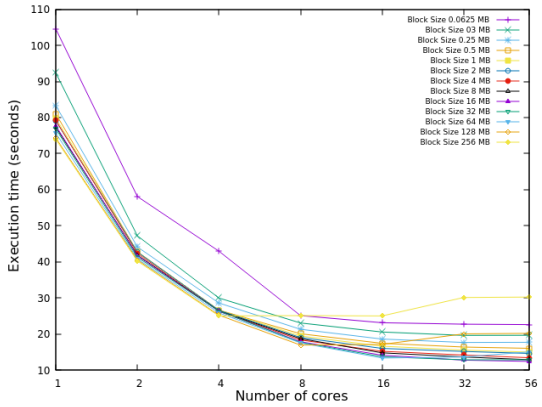


(a) Bandwidth respecte al número de *cores* per a diverses mides de bloc

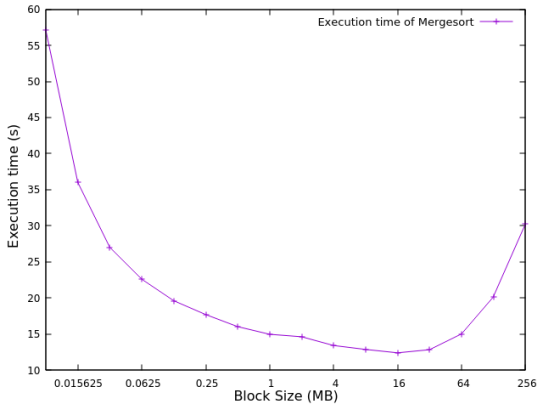


(b) Bandwidth respecte a la mida de bloc utilitzant 56 *cores*

Figura 25: *Bandwidth* obtingut en funció del número de cores i en funció de la mida de bloc per un fitxer d'1GB



(a) Temps total respecte al número de *cores* per a diverses mides de bloc



(b) Temps total respecte a la mida de bloc utilitzant 56 *cores*

Figura 26: Temps total mesurat en funció del número de cores i en funció de la mida de bloc per un fitxer d'1GB

Podem veure amb més detall el comportament de l'algoritme analitzant la traça d'Extrae que apareix a la figura 27. Podem veure com, un cop s'han ordenat els diferents blocs, de forma paral·lela, les primeres fases de *merge* (franges de color vermell envoltades de franges verdes i rosa) es poden executar de forma paral·lela. A mesura que pugem nivells de l'arbre de recursivitat, però, la mida de les operacions de *merge* va augmentant i, com que la operació de *merge* és seqüencial, s'acaba creant un desequilibri de la càrrega de treball, ja que la majoria de *threads* no estan ocupats. Aquesta característica inherent de l'algoritme limita el potencial paral·lelisme màxim que es pot aconseguir, tot i que la fàcil descomposició que permet en tasques el fa un candidat ideal per mostrar l'ús de TASPDK i la seva integració amb les directives d'OmpSs-2.

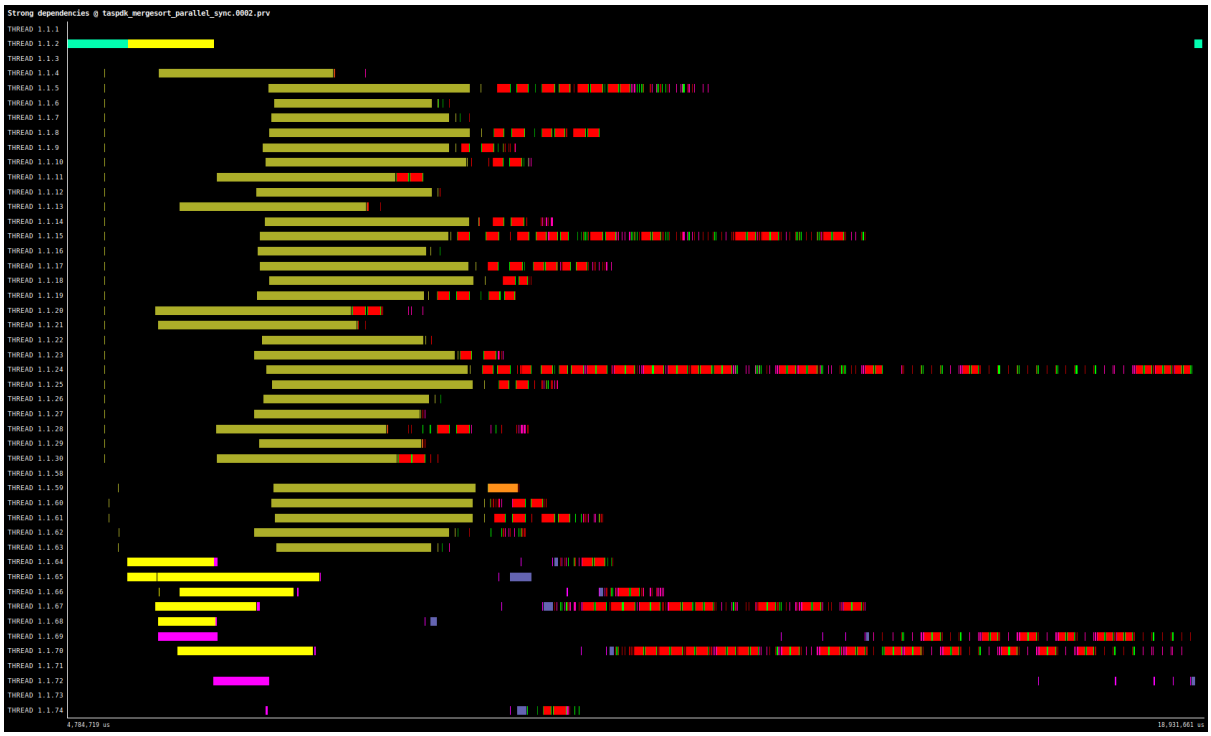


Figura 27: Traça d'Extrae d'una execució de l'*External Mergesort*

9.4 *K-Means*

L'algoritme de *k-means* és un mètode de *clustering* molt utilitzat per tal de classificar dades. L'objectiu del *k-means* és particionar n elements entre k clústers de forma que cada element pertany al clúster que té més proper, minimitzant la variància dins de cada clúster.

A la figura 28 apareix una visualització d'un *clustering* de 256 punts en el pla [-1.0, 1.0] en 5 clústers extret de la implementació que s'ha realitzat per aquest treball.

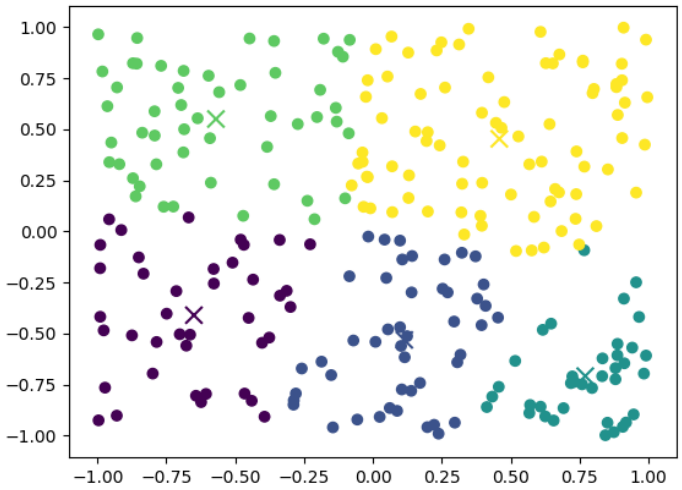


Figura 28: Visualització d'un *k-means clustering* de 256 punts en 5 clústers

Donada un set de mostres $m_1, .., m_n$ que es volen particionar en k clústers, l'algoritme de *k-means* funciona alternant dos passos fins que s'arribi a la convergència:

- **Assignació**: Cada mostra m_i s'assigna al label L_p, corresponent al clúster k_p més proper:

$$L_p = \{m_i : euclidean\_distance(m_i, k_p) \leq euclidean\_distance(m_i, k_j) \forall j, 1 \leq k\}$$

- **Actualització dels centres**: Es mou el centre de cada clúster tal que el nou centre és la mitjana de les mostres que composen el clúster:

$$k_i = \frac{1}{|L_i|} \sum_{m \in L_i} m$$

Aquests passos es repeteixen fins que s'arriba a una convergència. Existeixen diferents criteris de convergència; en aquesta implementació s'ha optat per aturar la simulació quan s'arriba a un màxim d'iteracions o quan la diferència de posicions entre els centres dels clústers abans i després de cada iteració se situa per sota d'un llindar *delta* configurable.

Cal destacar que abans d'iniciar el *clustering* s'ha de de decidir el centre inicial de cada un dels k clústers. Existeixen diferents mètodes per fer aquesta inicialització. El més simple, i l'utilitzat en aquesta implementació, consisteix en agafar k punts aleatoris dins del domini de les mostres, i fer-los servir com als centres inicials. Si aquests acabessin estant molt aprop entre ells, el *clustering* que s'obtindria no seria òptim. En general doncs, no es pot assegurar que el

clustering mitjançant *k-means* convergeixi sobre un mínim global, sinó un de local, que podria ser arbitràriament pitjor que la solució global, i sovint és necessari realitzar múltiples execucions i calcular una mitjana de tots els resultats per trobar una solució suficientment òptima.

Per tal de paliar aquesta problemàtica existeixen altres mètodes d'inicialització, com el *k-means++*, que asseguren una distribució més uniforme dels centres inicials.

Per tal de poder fer el *clustering* d'un *dataset* de mida M disposant d'una memòria de mida m, a cada iteració de la fase d'assignació s'han defectuar com a mínim $\frac{M}{m}$ lectures al disc. Addicionalment, com que la fase d'assignació es pot executar en paral·lel, es pot dividir la memòria en n blocs de mida BS tal que $m = n * BS$. S'ha de trobar una mida adequada de BS que permeti explotar al màxim el paral·lelisme sense crear un *overhead* massa gran degut al número de tasques o número d'operacions d'E/S. A la figura 29 apareix un dibuix amb el graf de tasques que es creen en cada iteració.

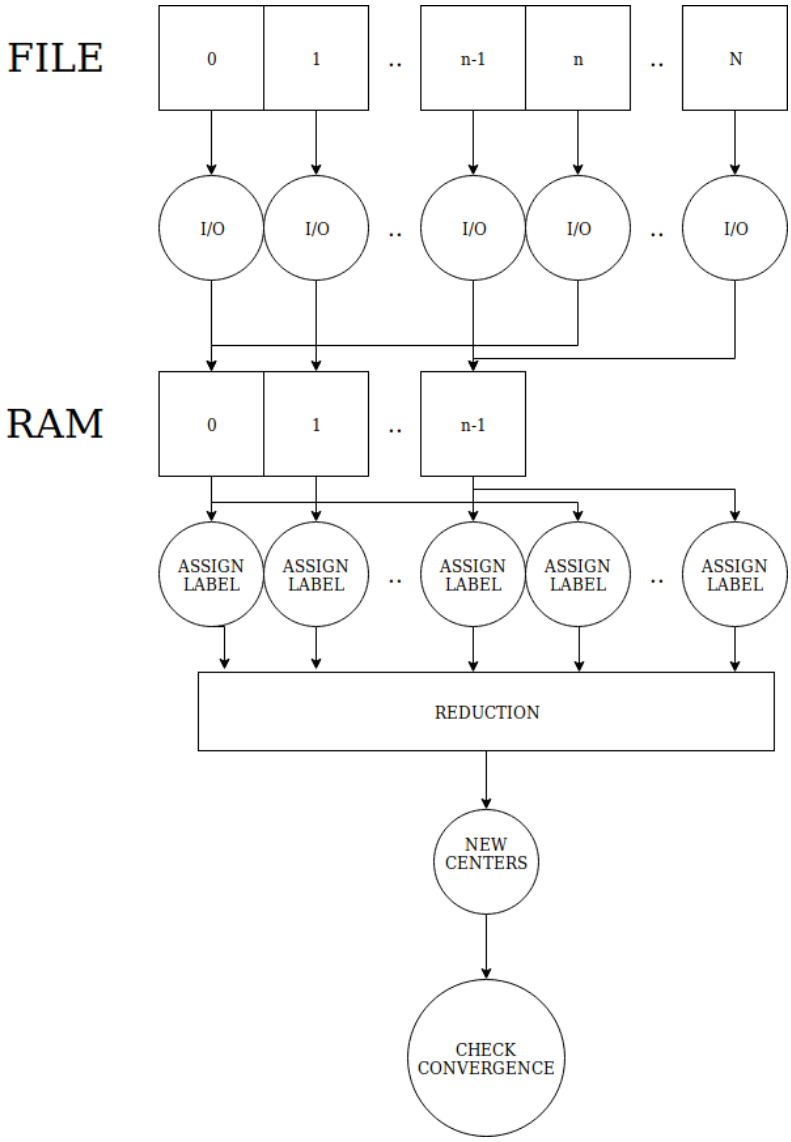


Figura 29: Graf de tasques creat en una iteració del *k-means*

A la figura 30 es pot veure el codi que s'encarrega de realitzar les reduccions descomposant en tasques el càlcul. S'ha usat el mecanisme de les reduccions d'arrays, suportat en OmpSs-2

des de la versió 18.06 [22]. Les reduccions permeten extreure paral·lelisme en una operació que altrament requeriria sincronització, sempre i quan aquesta compleixi certes propietats. En aquest cas, a l'hora de fer el recompte del número de mostres que s'assignen a cada clúster, s'hauria de serialitzar totes les tasques ja que totes requereixen accedir al vector amb una dependència *inout*. Gràcies a les reduccions, però, es pot realitzar cada suma parcial de forma privada a cada tasca i computar el resultat final un cop hagin acabat totes. Cal destacar que s'ha mogut la operació de lectura a una tasca separada ja que actualment no està suportat l'ús de les APIs de pausa de tasques ni d'events externs dins d'una tasca que realitza una reducció.

```
for(size_t k=0; k < FILE_SIZE; k+= TASPDK_RAM_SIZE){
       for(size_t i=0; i<TASPDK_RAM_SIZE; i+= PER_TASK_SIZE){
           struct kmeans_sample *samples = (struct kmeans_sample *)(buf+i);
           char *file_ptr_offset = (char *)file_ptr + k + i;

           #pragma oss task out(samples[0;PER_TASK_SAMPLES]) firstprivate(i,k) in(
               file_ptr_offset[0; PER_TASK_SIZE]) label(read_samples)
           {
               while(taspdk_read(samples_fd, samples, i + k, PER_TASK_SIZE)<0);
           }

           #pragma oss task reduction(+: [dataset->n_clusters]new_clusters_size) \
           reduction(+: [dataset->n_clusters]new_clusters_mean_x) \
           reduction(+: [dataset->n_clusters]new_clusters_mean_y)\
           in(samples[0;PER_TASK_SAMPLES]) \
           in(dataset->centers[0;dataset->n_clusters]) \
           firstprivate(samples) label(find_closest_center)
           {
               for(int j=0; j<PER_TASK_SAMPLES; j++){
                   int min_center = find_closest_center(samples[j].coords, dataset
                       ->centers, dataset->n_clusters);
                   assert(min_center < dataset->n_clusters);

                   new_clusters_size[min_center]++;
                   new_clusters_mean_x[min_center] += samples[j].coords.x;
                   new_clusters_mean_y[min_center] += samples[j].coords.y;
               }
           }
       }
   }
```

Figura 30: Codi que s'encarrega de la fase d'assignació

Totes les execucions del *k-means* implementat que es mesuren a continuació han estat executades utilitzant un *dataset* d'entrada de 8GB i reservant 2GB de memòria per operar amb les dades.

9.4.1 *Strong Scalability*

A la figura 31 apareix la *strong scalability* que presenta la implementació del *k-means* respecte al número de processadors utilitzats, per a diverses mides de bloc. La versió que utilitza l'API asíncrona s'ha omès per claredat, ja que no presenta cap diferència de rendiment respecte la versió síncrona. Com es pot observar, la implementació no presenta un *speedup* addicional a partir de 16 cores. Això és degut a que, tal i com es pot veure a la figura 32 a), ja s'està saturant el disc NVMe, i els cores adicionals no disposen de dades per treballar.

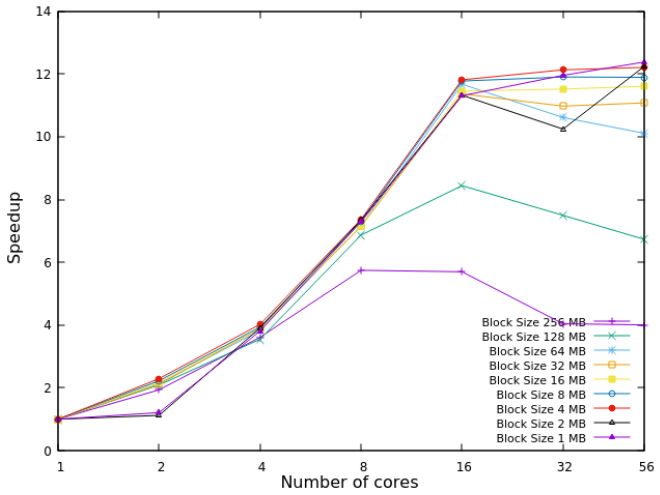
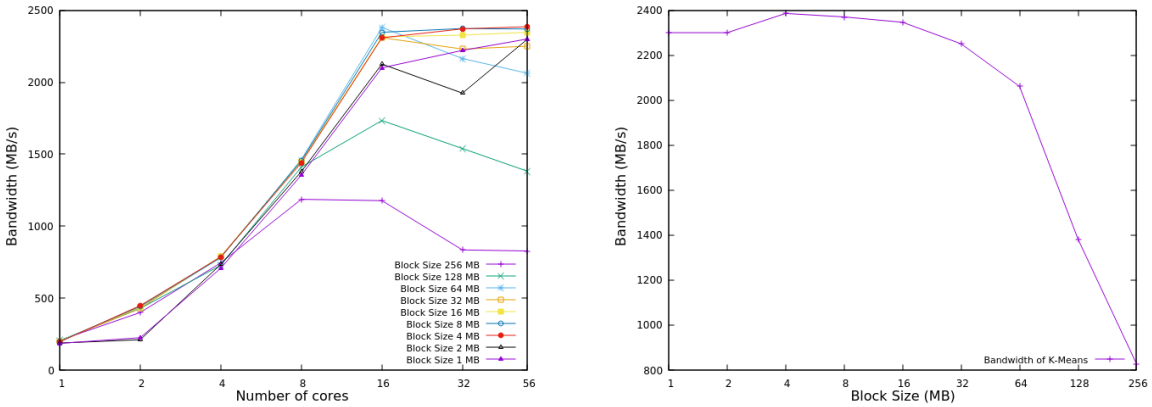


Figura 31: *Speedup* obtingut per a diferents mides de bloc



(a) *Bandwidth* aconseguit respecte al número de *cores* per a diferents mides de bloc en el K-Means

(b) *Bandwidth* aconseguit respecte a la mida de bloc utilitzant 56 *cores* en el K-Means

Figura 32: *Bandwidth* aconseguit per a diferents mides de bloc i número de *cores*

Una mesura més adequada per evaluar l'escalabilitat de l'algoritme sense tenir en compte el coll d'ampolla del disc és mesurar l'*speedup* aconseguit en la part computacional més important, que és la etapa d'assignació, ja que és totalment paral·lelitzable; cada mostra es pot tractar independentment. Per a fer-ho s'ha instrumentat el codi per trobar el temps destinat a cada execució a la fase d'assignació de les etiquetes.

Com es pot comprovar a la figura 33, a mesura que augmenta el número de *cores*, la fracció del temps total que correspon a la part d'assignació es redueix, i comença a contribuïr més en el temps total altres factors com el *bandwidth* del disc o la sincronització entre les diferents tasques. A la figura 34 apareix l'*speedup* aconseguit respecte el número de *cores* només en la fase d'assignació, i podem comprovar que aquest fragment presenta una escalabilitat molt superior a la de l'algoritme en conjunt.

Un altre factor que podria limitar la escalabilitat és el fet que, en el bucle principal, després de cada iteració s'ha de comprovar si s'ha convergit. Si es coloca una directiva *taskwait* després de cada iteració, les tasques creades es destrueixen i es tornen a crear a continuació, cosa que afegeix un *overhead* significant. En aquesta implementació s'ha obtat per un *unrolling*: Es llancen les dos primeres iteracions de forma consecutiva, i es comprova la convergència de la

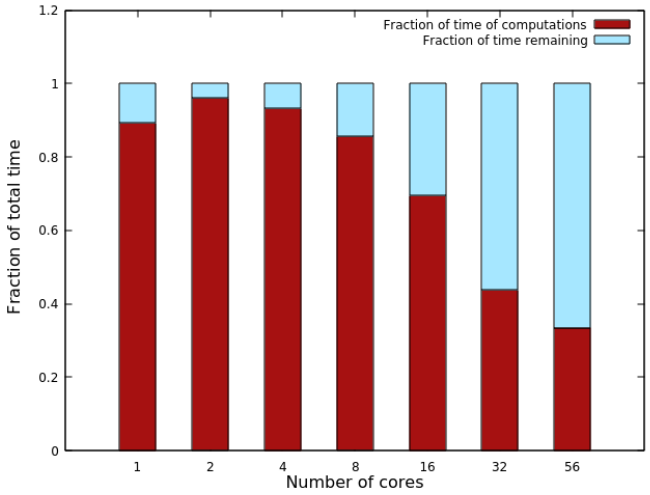


Figura 33: Fracció de temps respecte al temps total de la part computacional. Execució amb una mida de bloc de 8MB

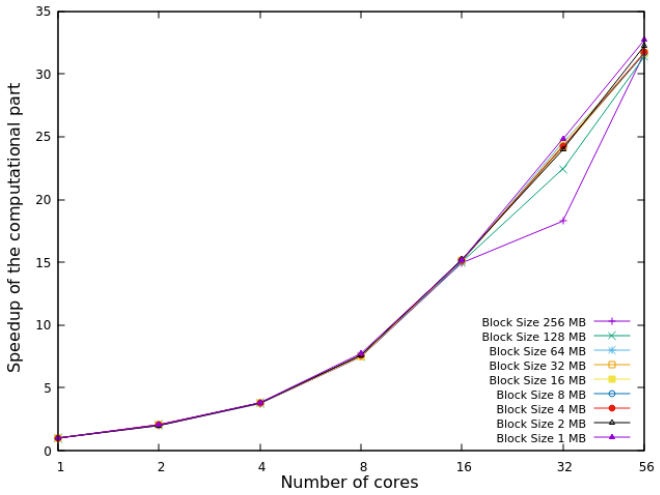


Figura 34: *Speedup* aconseguit en la part computacional per diferents mides de bloc

```
do {
    #pragma oss task out(old_convergence) in(convergence)
    old_convergence = convergence;

    ..

    /*Fase de còmput omesa */

    #pragma oss task out(convergence) in(dataset->centers[0;dataset->n_clusters]) \
    in(old_centers[0;dataset->n_clusters]) inout(iteration) label(convergence)
    {
        iteration++;
        convergence=check_converged(old_centers, dataset, iteration, &old_diff);
    }

    #pragma oss taskwait in(old_convergence) label(iteracions)
} while(!old_convergence);
```

Figura 35: Comprovació de la convergència a través d'*unrolling*

iteració *i* abans de començar a executar la iteració *i+2*. D'aquesta forma, la iteració *i+1* pot estar preparada esperant que es lliberin les dependències de la iteració *i* i començar a executar-se de seguida que això passi. A la figura 35 es mostra la implementació d'aquesta estratègia. La primera tasca creada es podrà executar immediatament ja que no existeix cap dependència de sortida sobre *converge* en aquesta iteració. S'executarà una iteració del algoritme fins arribar a la directiva *taskwait* sobre la variable *old_ convergence*. Com que la primera tasca, que presentava una dependència de sortida sobre aquesta variable, ja s'haurà executat, es farà una altra iteració del bucle. Quan s'arribi a la primera tasca de la segona iteració, es guardarà a *old_ convergence* el resultat de comprovar la convergència a la primera iteració i executarà els fragments de còmput de la segona iteració que ja no presentin dependències respecte a la prmiera iteració. Això permet que les tasques no es creïn i es destrueixin a cada iteració, si no que sempre estiguin pendents d'executar-se per quan s'alliberin les dependències. Finalment, al final d'aquesta iteració la directiva *taskwait* espera al resultat de la iteració anterior. Si hagués convergit, s'aturaria el bucle. Cal observar que fent servir aquest mètode s'efectua una iteració més de les necessàries.

A la figura 36 i 37 es pot veure les diferències entre dues execucions utilitzant una directiva taskwait vers una estratègia d'*unrolling*. Es pot comprovar com, utilitzant una estratègia d'*unrolling*, quan la segona iteració passa a executar-se perquè s'han satisfet les dependències sobre els buffers que emmagatzemen els nous centres dels clústers, moltes tasques que executen l'etapa d'assignació poden passar a executar-se immediatament ja que les lectures de les dades que necessiten s'han produit mentre s'esperava que acabessin totes les tasques de la iteració anterior. Així doncs, la segona iteració tarda menys temps en executar-se que la primera. No es així en la traça que utilitza un taskwait, ja que les tasques no poden avançar les lectures de la següent iteració encara que les dependències sobre els *buffers* que contenen les mostres estiguin satisfetes degut a la sincronització imposada pel *taskwait*. Això fa que, per exemple, totes les iteracions tinguin aproximadament la mateixa durada.

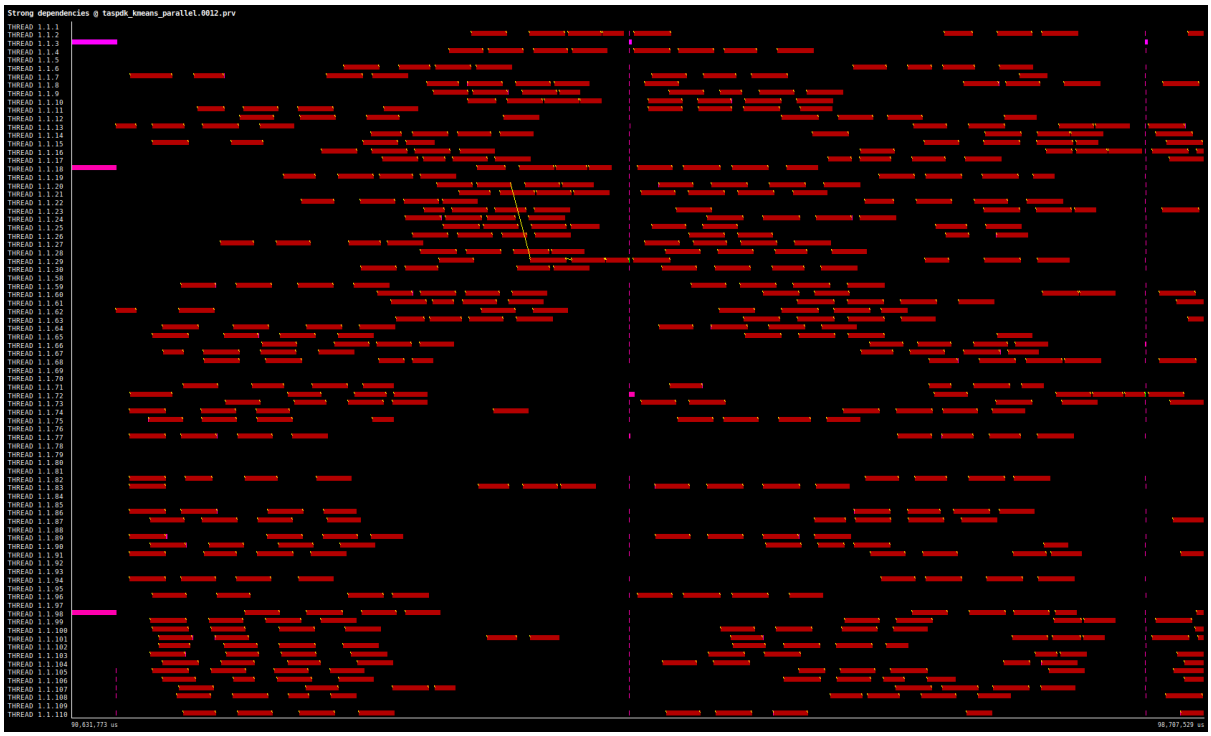


Figura 36: Iteracions del *k-means* utilitzant un *taskwait*

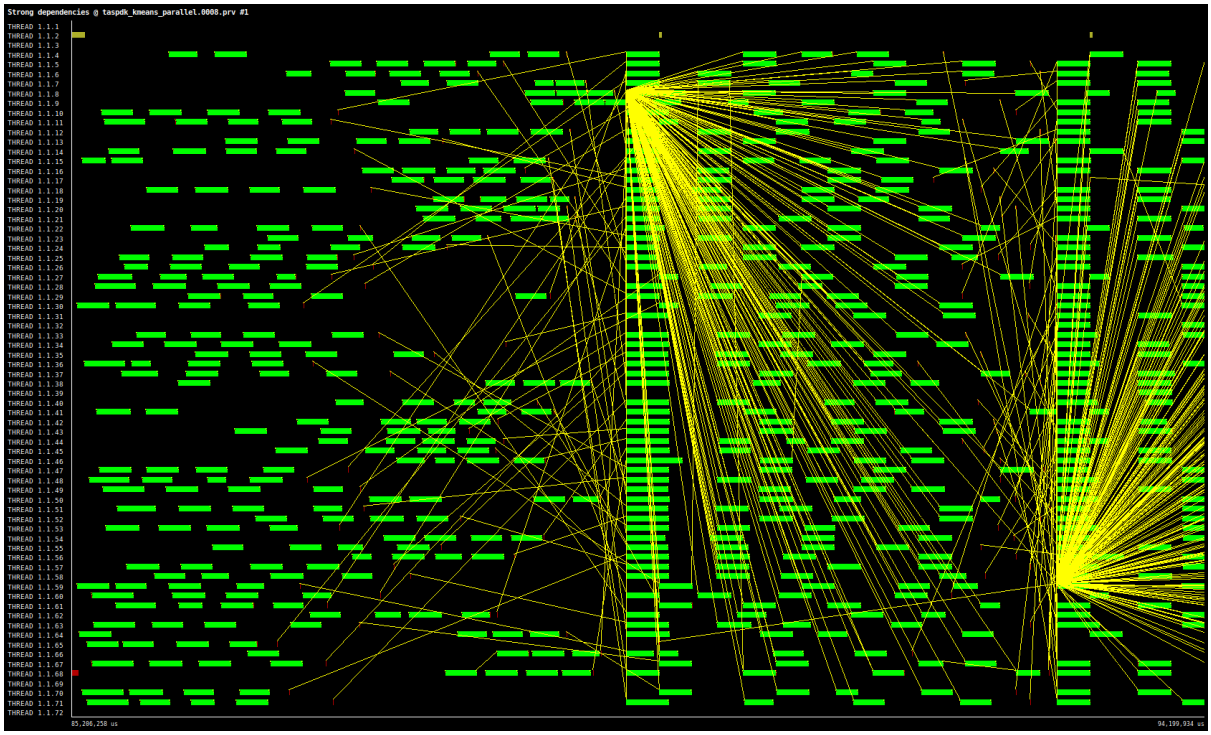


Figura 37: Iteracions del *k-means* utilitzant l'*unrolling*

9.5 *Convolutional Image Filter*

Una tècnica molt usada en el processament digital d'imatges és l'anomenat filtre de convolució. Aquest filtre consisteix en calcular el valor de cada píxel de la imatge com la suma dels seus veïns multiplicats per uns coeficients. Aquests coeficients s'especifiquen mitjançant una matriu, de mida $k * k$, essent k un número imparell, anomenada *kernel*. Per cada píxel s'efectua una operació de convolució entre dues matrius, la matriu de mida $k * k$ formada per el píxel a tractar al centre i els seus veïns i el *kernel*. Per exemple, donada la següent matriu i el següent kernel:

$$m = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, k = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

El valor del píxel e serà igual a

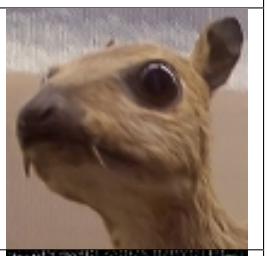
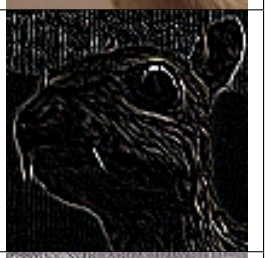
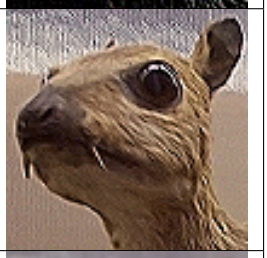
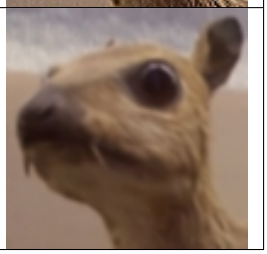
$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} [2,2] = (i*1)+(h*2)+(g*3)+(f*4)+(e*5)+(d*6)+(c*7)+(b*8)+(a*9)$$

Cal observar que per per la operació de convolució s'inverteixen les files i les columnes del kernel. En cas contrari s'està efectuant una correlació, que presenta diferents propietats matemàtiques, per exemple, no és additiva mentre que la convolució si.

Existeixen una sèrie de *kernels* que se solen utilitzar per determinades aplicacions. A la taula 5 apareixen una sèrie de *kernels* usats comuntment i el seu efecte sobre una imatge.

En molts àmbits es necessita aplicar filtres a imatges de mida molt gran, per exemple si es tracten d'imatges de satèl·lit per tal de filtrar soroll o intentar extreure més característiques de la imatge. En aquest cas, una estratègia que es pot seguir és descomposar la imatge en blocs que càpiguen en memòria RAM i aplicar la convolució per separat a cada un dels blocs. A més a més, la operació de convolució es pot aplicar en paral·lel dividing cada bloc en files que es poden calcular per separat. En definitiva, l'algoritme és del tipus *embarrasingly parallel*. A la figura 38 apareix un diagrama amb la descomposició en tasques descrita anteriorment.

Tal com s'acaba de comentar, la operació de convolució es pot aplicar de forma independent a cada píxel de la imatge, per tant és una candidata ideal a ser descomposada en tasques d'OmpSs-2. La estratègia que s'ha optat per seguir és la de dividir cada bloc de BSx files en n tasques

Operació	*kernel*	Resultat
Identitat	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge Detection	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Gaussian Blur 3x3	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Taula 5: Diferents *kernels* i els seus efectes sobre una imatge. Imatges per Michael Plotke sota llicència CC BY-SA 3.0

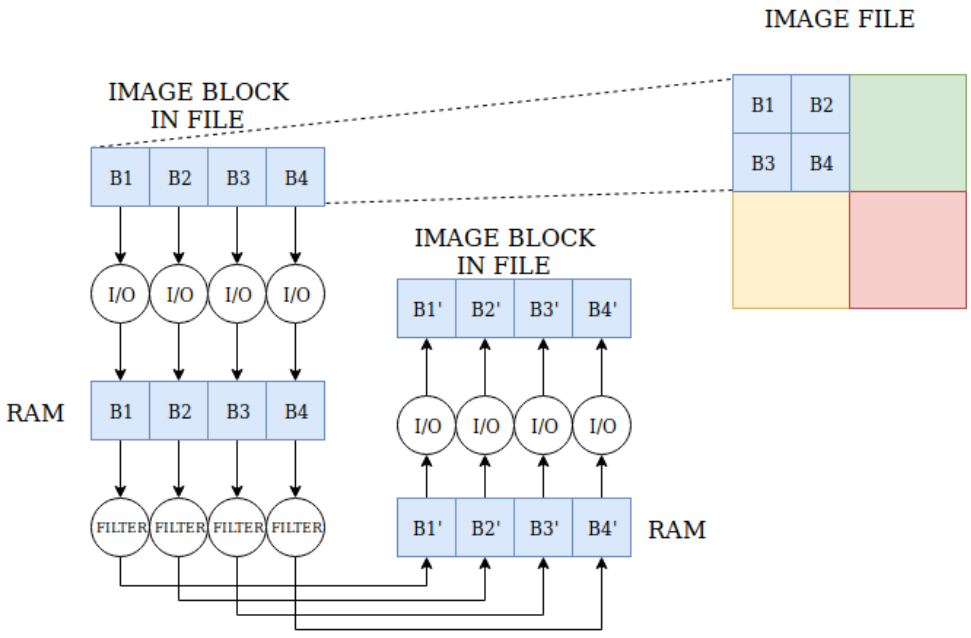


Figura 38: Graf de tasques creat per filtrar un bloc de la imatge.

de forma que cada tasca s'encarrega de processar $\frac{BSx}{n}$ files. A la figura 39 apareix el codi que implementa la convolució al qual s'han afegit les directives d'OmpSs-2 per paral·lelitzar-lo en tasques.

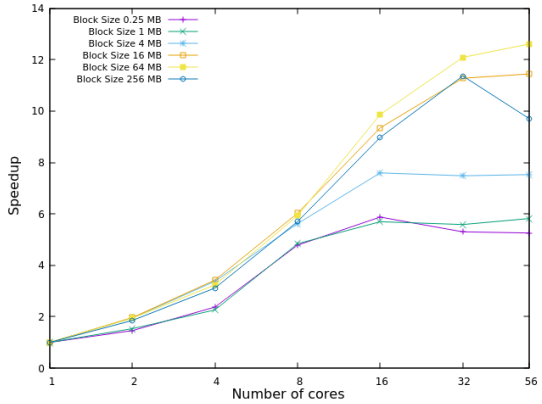
```
void filter_block(unsigned char *block_input, unsigned char *block_output, float (*
    kernel)[3])
{
    size_t rows_per_task;
    for(int i=0; i<BSx; i+=rows_per_task){
        #pragma oss task firstprivate(i) in(block_input[i*BSx;BSx*rows_per_task]) out(
            block_output[i*BSx;rows_per_task*BSx]) label(inner_filter)
        {
            for(int it=i; it<(i+rows_per_task); it++){
                for(int j=0; j<BSx; j++){
                    float pixel_value = 0.0f;
                    if(it > 0 && it<BSx-1 && j>0 && j<BSx-1){
                        for(int ii=0; ii<KS; ii++){
                            int block_pos_i = it+(ii-1);
                            for(int jj=0; jj<KS; jj++){
                                int block_pos_j = j+(jj-1);
                                pixel_value += ((float)block_input[INDEX_XY(block_pos_i,
                                    block_pos_j, BSx)])*kernel[ii][jj];
                            }
                        }
                        if(pixel_value < 0.0f) pixel_value = 0.0f;
                        if(pixel_value > 255.0f) pixel_value = 255.0f;
                    }
                    else{
                        pixel_value = ((float)block_input[INDEX_XY(it, j, BSx)]);
                    }
                    block_output[INDEX_XY(it,j,BSx)] = (unsigned char)pixel_value;
                }
            }
        }
    }
}
```

Figura 39: Codi que s'encarrega d'implementar la convolució paral·lelitzat mitjançant tasques

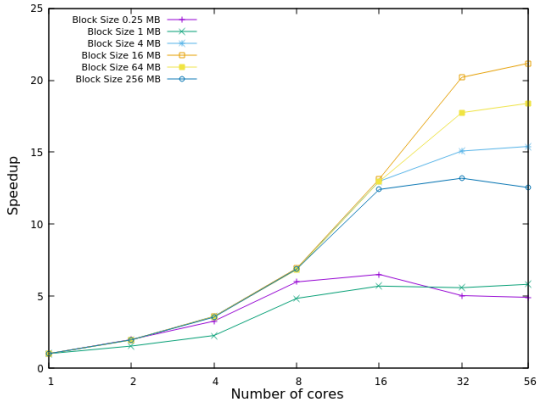
Per tal d'efectuar les següents proves de rendiment s'ha utilitzat una imatge per satèl·lit d'Europa publicada per la NASA al catàleg Visible Earth, que ocupa una mida d'1GB. Totes les proves s'han executat amb 512MB de memòria disponibles per operar amb les dades.

En aquest programa de prova, a diferència dels programes descrits anteriorment, hi ha una diferència substancial de rendiment entre les versions síncrones i asíncrones, així que es donaran els resultats per ambdues versions.

9.5.1 *Strong Scalability*



(a) Speedup obtingut per a diferents mides de bloc utilitzant les crides síncrones



(b) *Speedup* obtingut per a diferents mides de bloc utilitzant les crides asíncrones

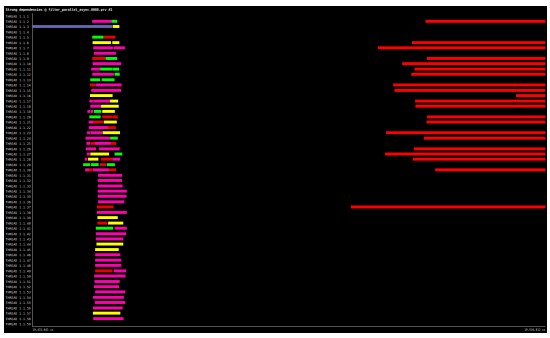
Figura 40: Speedup utilitzant diferents mides de bloc

A la figura 40 (a) apareix la *strong scalability* que presenta la implementació del filtre de convolució utilitzant les crídes sincrones respecte a diferents mides de bloc, i a la 40 (b), per la versió asíncrona. Com podem observar en aquest programa de prova, sí que s'obté una millora de rendiment quan s'utilitzen les crides asíncrones, un *speedup* màxim de 12 en el programa síncron contra un *speedup* màxim de 21 en el cas de l'asíncron. En el cas de la versió asíncrona, s'obté un *speedup* gairebé lineal fins a 16 *cores*.

Una possible explicació d'aquest fet és la forma en que es realitzen les peticions a disc. A la figura 42 es mostra el codi que carrega el bloc especificat per les coordenades *i,j* a un *buffer*; les escriptures estan estructurades de manera idèntica. Com que crear una tasca per la lectura de cada fila implicaria un número de tasques excessiu, es divideix el bloc utilitzant el paràmetre *NUMBER_OF_TASKS_PER_BLOCK*. El número de files que llegirà cada tasca és , per tant, elevat, i cada fila requereix un accés per separat ja que no estan guardades de forma seqüencial al disc. Si s'utilitza les crides síncrones, la tasca es bloquejarà i es desbloquejarà a cada operació, i el fet que cada tasca operi sobre un número relativament elevat de files fa que aquest fet introdueixi un *overhead* important. En canvi, les crides asíncrones podran fer totes les peticions seguides i després el *thread* queda lliure per fer-ne d'addicionals. Com s'ha comentat, es llegeixen files del disc que no estan guardades de forma contigua; aquest comportament no es donava en els altres programes de prova, ja que no apareixien tasques que consistissin a executar un bucle de peticions a disc saltant un *offset* en cada iteració, es feien peticions de la mida necessària a dades que estaven guardades de forma contigua. Això explicaria que els altres programes no presentessin diferències a l'utilitzar les crides asíncrones.

Si analitzem una traça d'Extrae de cada versió, podem veure que quadra amb la hipòtesi que s'ha formulat. A la figura 41 a) i a la figura 41 b) apareix una traça de la primera fase de càrrega de blocs i el conseqüent filtratge aplicat a aquest bloc per a la versió asíncrona i per a la versió síncrona, respectivament. En el cas de la versió asíncrona, al principi de la fase de càrrega de blocs s'efectuen totes les peticions al disc, que apareixen a la traça de color rosa.

Com que les crides són asíncrones, la tasca no s'ha d'esperar a que acabi l'anterior per a fer la següent. Quan totes les peticions corresponents al bloc s'hagin completat, començarà la fase de filtratge, que apareix al gràfic de color vermell. Per contra, en la versió asíncrona, la càrrega d'un bloc a memòria implica suspendre la tasca, que apareix a la traça com a discontinuitats entre cada lectura, i tornar-la a reprendre per a cada fila que es llegeixi. Això introdueix un *overhead* important que limita el rendiment del programa de forma significativa.



(a) Fase de lectura d'una execució utilitzant les crides asíncrones

(b) Fase de lectura d'una execució utilitzant les crides síncrones

Figura 41: Traces d'Extrae d'una execució utilitzant les crides síncrones

```
void load_block_from_image(taspdk_fd image, unsigned char *block, unsigned int block_i
    ,
    unsigned int block_j)
{
    size_t image_block_offset = (block_i*Nx*BSx)+(block_j*BSx);
    void *image_ptr = taspdk_addr(image);

    unsigned int ROWS_PER_TASK = BSx/NUMBER_OF_TASKS_PER_BLOCK;

    for(int k=0; k<NUMBER_OF_TASKS_PER_BLOCK; k++){
        size_t task_block_offset = ROWS_PER_TASK*k*BSx;
        size_t task_image_offset = ROWS_PER_TASK*k*Nx;
        #pragma oss task out(block[task_block_offset;ROWS_PER_TASK*BSx]) \
        firstprivate(task_block_offset,image_block_offset, task_image_offset) \
        label(read_rows_inner)
        {
            for(int i=0; i<ROWS_PER_TASK; i++){
                taspdk_aread(image, block+task_block_offset+(i*BSx), image_block_offset
                    +task_image_offset+(i*Nx), BSx);
            }
        }
    }
}
```

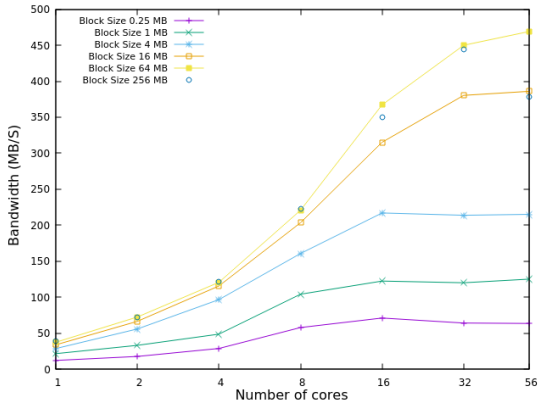
Figura 42: Codi encarregat de paral·lelitzar la lectura de fragments de la imatge des de disc

9.5.2 *Bandwidth*

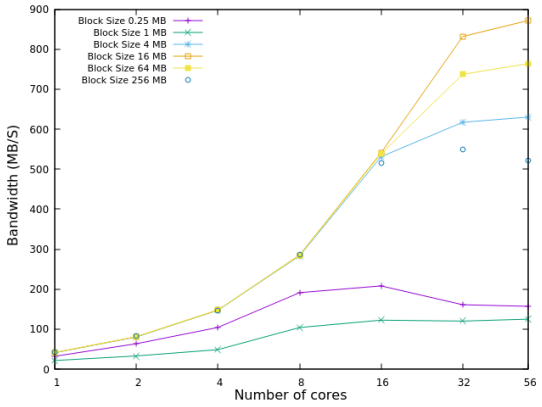
A la figura 43 (a) apareix el *bandwidth* que ofereix la implementació del filtre de convolució utilitzant les crídes sincrones respecte a diferents mides de bloc, i a la 43 (b), per la versió asíncrona. Igual que en el cas anterior, la versió asíncrona presenta un rendiment superior, oferint

gairebé el doble de *Bandwidth* que la versió síncrona. Aquests experiments també suporten la hipòtesi que s'ha plantejat a l'anàlisi de la escalabilitat. Si analitzem el *plot* del *bandwidth* respecte a la mida de bloc que apareix a la figura 45, podem veure com torna a aparèixer la gràfica en forma d'U invertida que hem vist anteriorment. En el cas de la versió síncrona, però, és menys pronunciada i fins a una mida de bloc de 256 MB escala de forma lineal. Això seria degut a que, tal com s'ha explicat anteriorment, l'*overhead* provocat per les crides síncrones dins de cada tasca introdueix una penalització important en el temps d'execució. Una mida de bloc més gran implica menys operacions al disc, ja que es poden fer menys accessos més grans, reduïnt l'impacte que provoca el fet d'utilitzar les crides asíncrones. Quan la mida de bloc és molt gran, però, es limita el paral·lelisme i això provoca una pèrdua de rendiment. A la figura 44 a) podem veure com, en el cas d'utilitzar les APIs síncrones, i donada la mida de bloc més petita que s'ha mesurat, la gran majoria del temps d'execució del programa no s'està executant la part de càlcul, degut a l'*overhead* introduït. Si s'utilitza les APIs asíncrones, la fracció del temps que s'està executant feina útil augmenta significativament, tal com es pot veure a la 44.

En aquesta prova, no s'aconsegueixen els *bandwidths* assolits en altres experiments. Possiblement aquest fet es deu a que les peticions al disc no es fan de forma seqüencial, com succeïa en els altres casos, si no que es tracta de peticions aleatories, que no ofereixen el mateix rendiment.



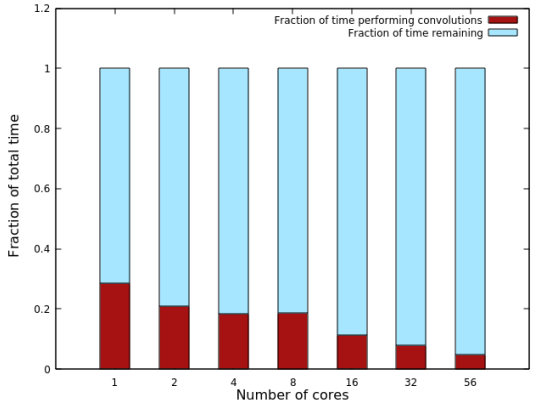
(a) Bandwidth obtingut per a diferents mides de bloc utilitzant les crides síncrones



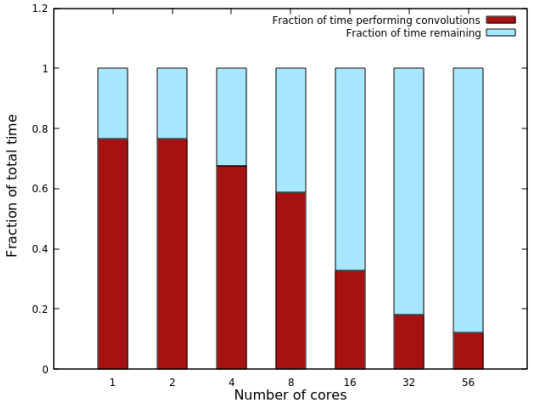
(b) *Bandwidth* obtingut per a diferents mides de bloc utilitzant les crides asíncrones

Figura 43: Bandwidth utilitzant diferents mides de bloc

A la figura 46 apareix l'*speedup* aconseguit només en el fragment de temps dedicat a la convolució per tal de no considerar el possible coll d'ampolla que pogués suposar el disc. En aquest cas no hi ha diferència de rendiment entre ambdues versions ja que la part que s'ha instrumentat no conté cap crida a disc. Fins a *16* cores, l'*speedup* és gairebé lineal, i comença a caure fins a arribar a un *speedup* màxim al voltant de 32 utilitzant 56 *cores*. La figura 47 mostra la eficiència obtinguda en la paral·lelització de la convolució. Com es pot veure, aquesta cau de forma pronunciada a partir dels 16 *cores*. La mida de bloc que filtra cada subtasca que forma tasca de la convolució coincideix amb la mida bloc que llegeix cada subtasca que forma la tasca de lectura o escriptura de blocs per tal que es puguin processar els blocs a mesura que es van llegint i, de forma anàloga, es puguin escriure al disc a mesura que es van filtrant. Cal destacar que, per limitacions de temps, no es va aprofundir en l'anàlisi dels paràmetres més òptims per la paral·lelització de la operació de convolució, i que la estratègia descrita anteriorment no sigui la més òptima, cosa que explicaria la caiguda sobtada que es dona a partir de l'ús de 16 *cores*.



(a) Fraccions del temps dedicat a la convolució en la versió síncrona



(b) Fraccions del temps dedicat a la convolució en la versió asíncrona

Figura 44: Anàlisi de la fracció de temps que es dedica a tasques de còmput utilitzant una mida de bloc de 0.25MB

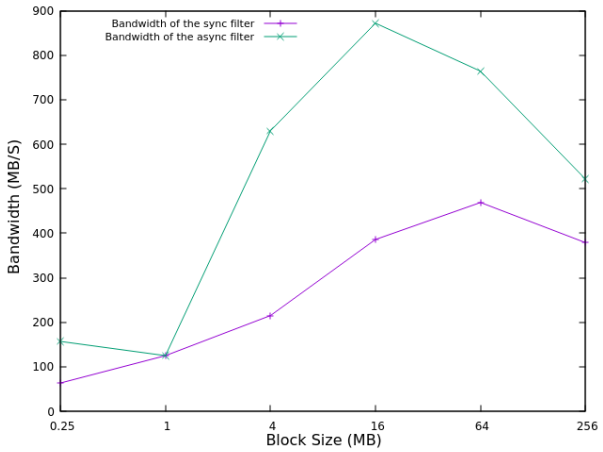
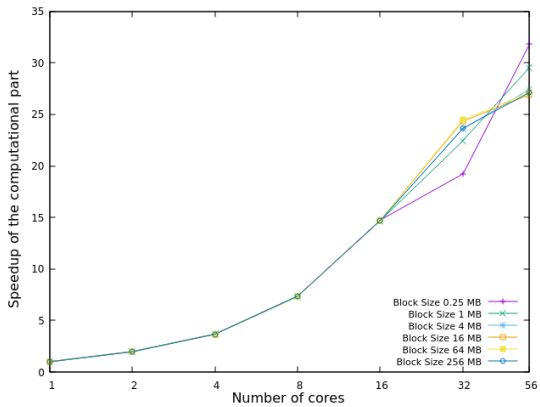
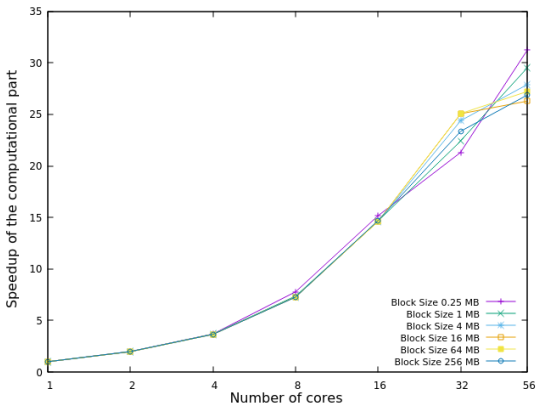


Figura 45: *Bandwidth* obtingut en funció de la mida de bloc per a 56 cores

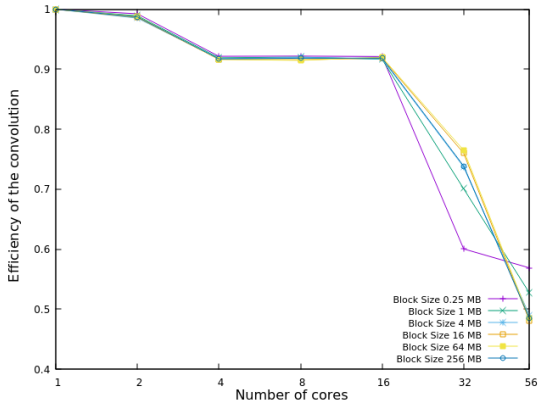


(a) Speedup obtingut en la convolució per a diferents mides de bloc utilitzant les crides síncrones

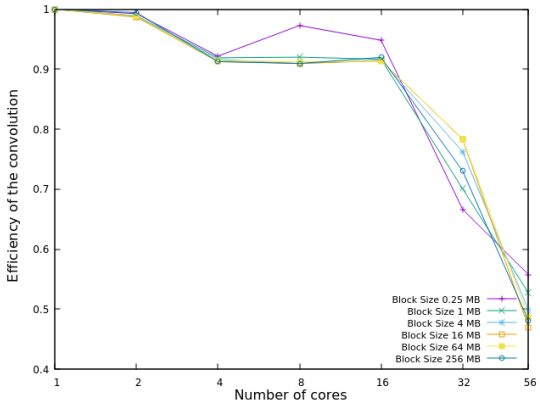


(b) *Bandwidth* obtingut en la convolució per a diferents mides de bloc utilitzant les crides asíncrones

Figura 46: *Speedup aconseguit en la convolució*



(a) Eficiència obtingut en la convolució per a diferents mides de bloc utilitzant les crides síncrones



(b) Eficiència obtingut en la convolució per a diferents mides de bloc utilitzant les crides asíncrones

Figura 47: Eficiència aconseguida en la convolució

10 Conclusions

Respecte al projecte, s'han assolit els objectius plantejats satisfactòriament. Primer, s'ha creat una llibreria que és capaç d'explotar al màxim l'elevat *bandwidth* que ens ofereixen els dispositius NVMe, introduïnt un *overhead* mínim respecte a d'SPDK. Segon, s'ha aconseguit una integració amb OmpSs-2 que ofereix una interfície al desenvolupador molt simple d'usar i que es mapeja de forma pràcticament transparent a les crides d'SPDK. Aquest últim punt és important, ja que estalvia molt temps de desenvolupament i de depuració potencial, que un programador hauria d'invertir si fes servir SPDK directament en les seves aplicacions. Sens dubte en el futur els dispositius NVMe guanyaran presència en entorns d'HPC, i aquesta llibreria presenta el primer pas per tal de poder explotar-ne el rendiment utilitzant el *programming model* d'OmpSs-2.

El projecte s'ha endarrerit respecte la previsió inicial, però s'ha complert a temps, si bé amb el temps just, gràcies al fet d'haver deixat marge suficient per cobrir imprevistos.

En l'àmbit personal he adquirit una sèrie de coneixements pràctics que van molt més enllà del que he vist al grau. Coneixements sobre el llenguatge C, explotar al màxim les eines integrades de Linux, conceptes generals de paral·lelisme i en particular d'OmpSs-2 i nanos6, i, sobretot, aprendre sobre estratègies i metodologies per a realitzar una depuració molt més eficient utilitzant eines com *gdb*, *valgrind* o *address sanitizer*.

11 Treball futur

De cara al futur seria interessant extendre la llibreria per tal de suportar més d'un dispositiu simultàniament. Aquest fet permetria explotar més paral·lelisme de les aplicacions ja que reduiria el coll d'ampolla existent distribuint el tràfic d'E/S entre més d'un disc.

També es podria utilitzar la especificació NVMe Over Fabrics (NVMe-OF) per tal d'explotar al màxim el rendiment de TASPDK en entorns distribuïts.

Finalment, es podria extendre el suport a altres *runtimes* o a OpenMP.

Referències

[1] Sanam Shahla Rizvi i Tae-Sun Chung. "Flash SSD vs HDD: High performance oriented modern embedded and multimedia storage systems". A: *2010 2nd International Conference on Computer Engineering and Technology*. Vol. 7. IEEE. 2010, pàg. V7-297.

[2] Ziye Yang et al. "Spdk: A development kit to build high performance storage applications". A: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pàg. 154-161.

[3] *OmpSs-2*. URL: `https://pm.bsc.es/ompss-2`.

[4] *Mercurium*. URL: `https://pm.bsc.es/mcxx`.

[5] *Nanos6*. URL: `https://github.com/bsc-pm/nanos6`.

[6] Kevin Marks. "An NVM Express Tutorial". A: *Flash Memory Summit* (2013).

[7] Benjamin Walker. "SPDK: Building blocks for scalable, high performance storage applications". A: *Storage Developer Conference. SNIA*. 2016.

[8] *Paraver*. URL: `https://tools.bsc.es/paraver`.

[9] *Extrae*. URL: `https://tools.bsc.es/extrae`.

[10] Brad Chamberlain. *Parallel Processing Languages: Cray's Chapel Programming*. Maig de 2013. URL: `http://www.cray.com/blog/chapel-productive-parallel-programming`.

[11] Amber Huffman i Sr Principal Engineer. "NVM express overview & ecosystem update". A: *Proceedings of Flash Memory Summit* (2013).

[12] Qiumin Xu et al. "Performance analysis of NVMe SSDs and their implication on real world databases". A: *Proceedings of the 8th ACM International Systems and Storage Conference*. ACM. 2015, pàg. 6.

[13] *What's New in Linux 3.3?* Març de 2012. URL: `https://www.linuxfoundation.org/blog/2012/03/whats-new-in-linux-3-3/`.

[14] Matias Bjørling et al. "Linux block IO: introducing multi-queue SSD access on multi-core systems". A: *Proceedings of the 6th international systems and storage conference*. ACM. 2013, pàg. 22.

[15] *Scaling Performance*. URL: `https://spdk.io/doc/nvme.html`.

[16] Ben Walker i Jim Harris. *10.39M Storage I/O Per Second From One Thread*. Maig de 2019. URL: `https://spdk.io/news/2019/05/06/nvme/`.

[17] Jisoo Yang, Dave B Minturn i Frank Hady. "When poll is better than interrupt." A: *FAST*. Vol. 12. 2012, pàg. 3-3.

[18] *Controlling Hardware From User Space*. URL: `https://spdk.io/doc/userspace.html`.

[19] *Direct Memory Access (DMA) From User Space*. URL: `https://spdk.io/doc/memory.html`.

[20] Ken Schwaber i Mike Beedle. *Agile software development with Scrum*. Vol. 1. Prentice Hall Upper Saddle River, 2002.

[21] Donald Lewine. *POSIX programmers guide*. "O'Reilly Media, Inc.", 1991.

[22] *New release of OmpSs-2*. Juny de 2018. URL: `http://www.intertwine-project.eu/news/2018/new-release-ompss-2-18-06-26`.

[23] Jeffrey Scott Vitter. *Algorithms and Data Structures for External Memory*. 2006.

Apèndixs

A Diagrama de Gantt

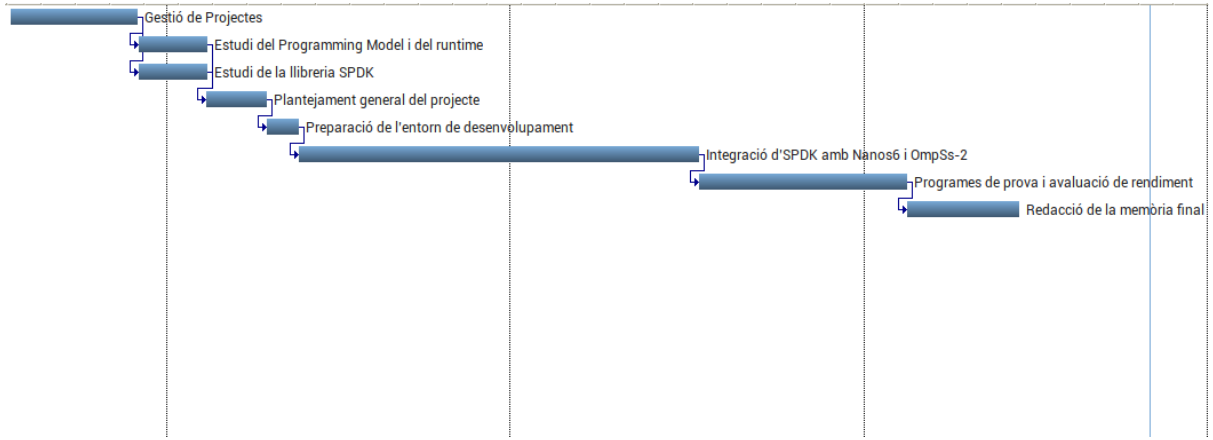


Figura 48: Diagrama de Gantt del Projecte